DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

UNIVERSIDAD AUTÓNOMA DE MADRID

Especificación, Análisis y Generación de Entornos para Lenguajes Visuales de Dominio Específico

Tesis Doctoral

Autor: Esther Guerra Sánchez

Directores: Dr. Juan de Lara Jaramillo
Dra. Paloma Díaz Pérez

2007

Resumen

Los Lenguajes Visuales de Dominio Específico (LVDEs) se utilizan con frecuencia para el análisis y diseño de sistemas, tareas que suelen realizarse en entornos de modelado. Para construir estos entornos la tendencia actual es partir de una definición de alto nivel del LVDE, desde la cual se genera una herramienta que permite definir modelos conforme a la sintaxis del lenguaje. Sin embargo, la complejidad de los sistemas software actuales requiere entornos más funcionales que permitan no sólo dibujar modelos, sino también verificar su consistencia, analizarlos, cuantificar su calidad, aplicar patrones de diseño, sintetizar código o generar informes.

Por esta razón, el objetivo de la presente tesis es facilitar la definición de un conjunto de funcionalidades orientadas al control de la calidad de los diseños, reduciendo o eliminando la necesidad de codificarlos. Para ello se propone un marco basado en técnicas visuales y formales para la definición de LVDEs que contempla la especificación de los tipos de diagramas propuestos por el lenguaje visual, la generación de mecanismos de consistencia sintáctica derivados automáticamente a partir de la definición del lenguaje, la verificación del sistema mediante su transformación a un dominio semántico formal y posterior anotación de resultados al lenguaje original, el uso de un lenguaje de consulta sobre modelos basado en patrones que proporciona sincronización entre el modelo consultado y el resultado de la consulta, la definición de métricas específicas de dominio, y la definición de rediseños específicos de dominio (automatizables a partir del valor de las métricas). La propuesta sigue un enfoque basado en meta-modelado para especificar la sintaxis del lenguaje y de los distintos tipos de diagrama, y en sistemas de transformación de grafos, patrones y otros modelos para especificar la funcionalidad adicional.

El marco propuesto se ha validado mediante su implementación en una herramienta de meta-modelado que se ha usado para generar entornos para LVDEs en diversos dominios.

Abstract

Domain Specific Visual Languages (DSVLs) are frequently used for the specification and design of systems. These tasks are usually automated by using modelling environments. There is a trend to build such environments by providing a high-level specification of the DSVL, and starting from this specification a tool is generated that allows defining models conforming to the language syntax. However, the complexity of current software systems requires more functional environments that allow not only drawing the models, but also verifying their consistency, analyzing them, quantifying their quality, applying design patterns, synthesizing code or generating reports.

In this way, the goal of this thesis is to facilitate the definition of a suite of functionalities oriented to controlling the quality of the designs, reducing or eliminating the necessity of coding. For this purpose, a framework for the definition of DSVLs is proposed that is based on visual and formal techniques. It includes techniques for the specification of the diagram types proposed by the visual language, the generation of syntactic consistency mechanisms derived from the language definition, the verification of the system by means of its translation into a formal semantic domain as well as the back-annotation of the analysis results to the initial notation, the use of a model query language based on patterns that provides synchronization between the base model and the model resulting from the query, the specification of domain specific metrics, and the definition of domain specific redesigns (possibly automated by metric values). The proposal follows a meta-modelling approach for the specification of the syntax and diagram types of the language, and on graph transformation systems, patterns and additional models to specify the extra functionality.

The proposed framework has been validated by means of its implementation on a meta-modelling tool that has been used to generate environments for DSVLs in several domains.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Los Lenguajes Visuales (LVs) se utilizan en ingeniería del software durante las fases de planificación, análisis y diseño con objeto de razonar sobre el sistema software en construcción. Existen estudios que demuestran cómo, en ocasiones, el uso de una representación visual facilita la comprensión de un sistema más que usar otra equivalente de carácter textual [90, 116]. Dependiendo de su ámbito de aplicación, los LVs se clasifican en dos categorías: de *propósito general* si proporcionan primitivas gráficas generales para modelar (potencialmente) cualquier sistema, y de *dominio específico* si están orientados a un área de conocimiento concreto para el que definen construcciones de alto nivel que representan conceptos del dominio. Los LVs de Dominio Específico (LVDEs) son fáciles de aprender porque manejan conceptos del área de aplicación, de modo que el salto semántico entre el modelo mental del usuario y el del sistema es menor. Por esta razón permiten incrementar la productividad de quienes los utilizan, así como la calidad de los sistemas especificados mediante su uso. Más aún, estos lenguajes son clave en paradigmas de desarrollo dirigido por modelos [188], donde los modelos desempeñan un papel fundamental ya que son el artefacto inicial a partir del cual se genera automáticamente toda (o parte de) la aplicación. Algunos dominios que han adoptado el uso de LVDEs son la ingeniería web [37, 53, 163], las bibliotecas digitales [126], el intercambio de datos electrónicos [81], el modelado de formalismos híbridos [111], la programación paralela [22] o el diseño de redes de tráfico [184].

Debido a la complejidad que están adquiriendo los sistemas software, una técnica habitual para facilitar su especificación es dividirla en varios diagramas que capturan una característica o vista del sistema utilizando el LV más apropiado (que puede ser de dominio específico). La familia de lenguajes usados para especificar un sistema se denomina *lenguaje multi-vista*. Un ejemplo de lenguaje multi-vista de propósito general es UML2.0, que define diferentes LVs para describir la estructura de un sistema (diagramas de clases y objetos) así como su comportamiento (diagrama de transición de estados, de comunicación, etc.). De este modo la especificación del sistema deja de ser monolítica, y pasa a estar formada por un conjunto de diagramas interrelacionados. Como consecuencia surge el riesgo de introducir inconsistencias sintácticas y semánticas entre los distintos diagramas que, no siendo detectadas a tiempo, lleven a la construcción de sistemas erróneos. Por ello se necesitan técnicas que ayuden a especificar formalmente estos lenguajes y que

proporcionen mecanismos para asegurar la consistencia entre diagramas, analizar el sistema descrito, y estudiar propiedades de interés. Tales mecanismos deberían especificarse a nivel de formalismo porque forman parte de la semántica (y por tanto de la definición) del lenguaje, y además porque permitiría hacerlos independientes de la tecnología usada para su implementación.

Además, junto a la corrección y compleción de las especificaciones, existen otros valores añadidos que pueden influir en el éxito de un proyecto software, como por ejemplo la facilidad de mantenimiento o uso del producto resultante. En este contexto, algunas de las técnicas usadas para estudiar y mejorar este tipo de propiedades de calidad son: las *métricas* para cuantificar las propiedades de un sistema [68], los *patrones de diseño* que aportan soluciones probadas a problemas de diseño recurrentes [76], y el *rediseño* de la estructura de los modelos de diseño con el fin de mejorar sus propiedades [105] (también llamado *refactoring* de modelos [74, 133] si preserva el comportamiento).

El control de todas estas propiedades determinantes de la calidad de un producto software debería incorporarse desde las fases iniciales del desarrollo [23], tarea para la cual disponer de herramientas apropiadas es fundamental. De lo expuesto anteriormente podemos concluir que los entornos usados para modelar sistemas deberían integrar mecanismos para: (i) verificar la consistencia y corrección sintáctica y semántica de los sistemas; (ii) obtener propiedades de interés; y (iii) cuantificar y mejorar su calidad. En paradigmas de desarrollo tradicionales, disponer de tales entornos puede ayudar a detectar defectos antes de la implementación, ahorrando tiempo y dinero. En paradigmas de desarrollo dirigido por modelos, donde se debe asegurar la corrección y calidad de los modelos antes de generar código, poder usar esos entornos es crítico para el éxito del proyecto.

Los *entornos de modelado* tradicionales (o herramientas CASE) permiten construir modelos escritos en un LV que puede ser multi-vista y de dominio específico, pero la definición del lenguaje es fija. Los entornos más avanzados proporcionan funcionalidades adicionales para la verificación, análisis, medición y rediseño de modelos. Sin embargo, usar estos entornos puede resultar difícil si no se está familiarizado con el lenguaje de modelado, o si éste no refleja los conceptos del dominio de aplicación. Además, los mecanismos de verificación y análisis suelen ser poco flexibles ya que vienen predeterminados y sin posibilidad de extensión. Por otro lado, los entornos existentes sólo cubren un pequeño grupo de las potenciales áreas de aplicación. Para el resto de áreas se requiere desarrollar nuevos entornos para el modelado con otros lenguajes. El problema es que desarrollar tales entornos para la amplia variedad de LVDEs (que a día de hoy siguen surgiendo) es una actividad que consume mucho tiempo y recursos, en especial si hay que integrar métodos formales de verificación y medición, siendo una tarea ardua que en muchos casos no se beneficia de desarrollos previos.

Para solucionar estos problemas surgieron las *herramientas de meta-modelado* [51, 55, 63, 78, 118, 136, 195], también llamadas herramientas metaCASE o generadores de edi-

tores de diagramas. Estas herramientas permiten generar automáticamente entornos de modelado para un LVDE dado. Para ello sólo se necesita especificar el lenguaje mediante un meta-modelo. Esto permite la construcción de entornos de modelado gráfico con gran rapidez, ya que los formalismos usados para definir el meta-modelo suelen ser gráficos y de alto nivel (como por ejemplo diagramas de clases UML). Sin embargo, los entornos así generados tienen menos funcionalidad añadida que los entornos de modelado tradicionales, ya que muchas veces funcionan como meros editores donde poder especificar, salvar y recuperar los modelos. Sólo en contadas ocasiones proporcionan mecanismos para la consistencia entre modelos [118, 136, 195], el análisis de propiedades o el soporte de criterios de calidad dentro del dominio tratado, mecanismos que a menudo tienen que codificarse a mano. Por tanto se necesitan nuevas técnicas y herramientas de meta-modelado que permitan generar entornos visuales con funcionalidades avanzadas a partir de una descripción formal y de alto nivel de las mismas.

1.1. Antecedentes

En la actualidad existen muchas herramientas de meta-modelado capaces de describir y generar entornos personalizados para LVDEs [51, 55, 63, 78, 118, 136, 195]. Sin embargo, algunas sólo permiten definir entornos para lenguajes con una única vista, es decir, formados por un solo tipo de diagrama [51, 55]. Para el caso (frecuente) de lenguajes multi-vista, la única posibilidad que ofrecen es generar una herramienta distinta e independiente para cada una de las vistas. Esto hace imposible garantizar la consistencia entre los modelos especificados en las distintas herramientas, y más aún su análisis.

Recientemente ha surgido una nueva familia de herramientas de meta-modelado que permite generar entornos visuales con soporte para la consistencia sintáctica y de la semántica estática entre las vistas de un sistema [63, 78, 118, 136, 195]. En algunos casos las reglas de consistencia entre vistas las especifica el diseñador del entorno visual [118], normalmente mediante código, lo cual es un trabajo tedioso, repetitivo y propenso a errores. En otros casos la herramienta genera automáticamente las reglas a partir de la información del meta-modelo [136, 195], prefijando el paradigma de comportamiento usado para mantener la consistencia en el entorno de modelado generado. Aunque esas reglas se puedan modificar posteriormente a mano, sería interesante poder seleccionar el paradigma de comportamiento antes de su generación, para de ese modo evitar manipulaciones que pueden dar origen a la introducción de errores inadvertidos. Además, esto daría al diseñador del entorno una mayor flexibilidad y control sobre el producto generado. En aquellas herramientas donde las reglas se generan automáticamente, el enfoque más usual para la consistencia se basa en construir un repositorio único donde las vistas del sistema están relacionadas. Algunas herramientas expresan esas relaciones de manera textual [152, 195],

lo que dificulta realizar cambios en el comportamiento que se genera por defecto en el entorno final. En otras ocasiones se utilizan notaciones parcialmente gráficas [82], lo que facilita su especificación. Sin embargo, pocas herramientas utilizan mecanismos gráficos, de alto nivel y formales para especificar esas relaciones.

Adicionalmente, algunas propiedades estáticas se pueden verificar informalmente realizando consultas sobre los modelos. Esto es muy útil para extraer información de modelos grandes o si el sistema está especificado con varios modelos. Casi todas las herramientas de meta-modelado disponen de algún lenguaje que permite realizar consultas sobre modelos, lenguaje que suelen heredar los entornos de modelado generados con ellas. Entre los lenguajes usados hay textuales [176] y visuales [173], así como declarativos [17, 173], imperativos e híbridos [155]. En algunos casos el resultado se muestra como un modelo nuevo, y en otros casos se devuelve como una lista de objetos lógicos [78, 176] que implica un salto de abstracción entre el modelo consultado y el resultado de la consulta. Por último, no es frecuente que los lenguajes proporcionen mecanismos de sincronización entre el modelo consultado y el resultante, de tal modo que para asegurarse de la consistencia del resultado hay que volver a realizar la consulta.

Respecto a mecanismos para garantizar la consistencia de la semántica dinámica (comportamiento), así como para el análisis de propiedades, el enfoque más frecuente es transformar los modelos a un formalismo común que proporcione los mecanismos adecuados para ello, ya sea mediante simulación o mediante técnicas de verificación formal [121, 184, 186, 192]. En algunas ocasiones el proceso de transformación se especifica de manera procedimental. En otros casos la transformación se especifica en herramientas de transformación de modelos que pueden estar integradas o no en la herramienta de meta-modelado. Si no están integradas, el diseñador del entorno visual está obligado a cambiar de herramienta según la tarea a realizar, y además debe especificar mecanismos para traducir los modelos a la notación utilizada por la herramienta de transformación (y viceversa). Posteriormente, en el entorno generado, el usuario también debe usar herramientas distintas para especificar y analizar el sistema. Si por el contrario, las herramientas de transformación están integradas en el entorno de meta-modelado, aún queda el problema de cómo mostrar los resultados del análisis al usuario. Idealmente, el entorno de modelado generado debería ser fácil de usar y orientado al usuario, lo cual significa que el proceso de transformación, análisis y verificación debería ser transparente al usuario, y las propiedades a verificar y las respuestas obtenidas se deberían expresar en el LVDE que el usuario está usando. Para ello, al mismo tiempo que el diseñador del entorno especifica la transformación del LVDE y los mecanismos de análisis, también debería definir cómo mostrar los resultados en términos del LVDE original. Sin embargo, casi ninguna herramienta de meta-modelado incluye mecanismos para ello. En la literatura existen muy pocas propuestas de mecanismos generales para la anotación de resultados [186]. Las escasas herramientas que mencionan mecanismos de este tipo [24] no explican cómo se realiza, o los mecanismos son poco flexibles, están

codificados sin posibilidad de ser modificados por el diseñador del entorno, y sólo permiten anotación uno-a-uno (un elemento del formalismo destino se traduce en un elemento del LVDE origen).

Por otro lado, como se indicaba en la introducción, una funcionalidad deseable en un entorno de modelado es la inclusión de procesos para medir y mejorar la calidad de los modelos, y en consecuencia del sistema que éstos representan. De hecho, la necesidad de nuevas herramientas para la modernización y evolución del software es un tema candente reconocido por la OMG. En la actualidad, su *Architecture-Driven Modernization Task Force* [3] está trabajando en el desarrollo de estándares para herramientas de modernización basadas en meta-datos, cuyo objetivo es facilitar el análisis, visualización, *refactoring* y transformación de los sistemas software existentes. Con este propósito han publicado una solicitud de propuestas (*Request for Proposals*) [4] para la especificación de un meta-modelo que defina métricas y *refactorings*, permita su intercambio, y sea lo suficientemente flexible para adoptar nuevas clases de métricas a un mínimo coste.

Hoy en día existe una gran variedad de entornos de modelado que incorporan funcionalidades para obtener métricas de los modelos [115, 164, 179]. Sin embargo, las métricas suelen estar codificadas a priori, no son modificables, y las posibilidades de extensión son muy limitadas. Algunas excepciones permiten extender el conjunto de métricas predefinido, pero para un lenguaje predeterminado [164]. En otros casos proporcionan métricas independientes del lenguaje que se pueden configurar para distintos lenguajes pero que, en cualquier caso, están siempre orientadas a un dominio concreto (muy frecuentemente el de la orientación a objetos) [115]. De manera similar, bastantes entornos de modelado incorporan capacidades de *refactoring*, aunque éstos suelen estar orientados a lenguajes específicos fijos, donde la detección de dónde aplicarlos debe hacerse a mano, y donde el conjunto de *refactorings* está predefinido y no es extensible por el usuario [157, 179]. Entre ellos, sólo un pequeño número permite detectar automáticamente oportunidades de *refactoring* [180].

En el área de herramientas de meta-modelado no existen ejemplos donde se proporcione un soporte real para definir de manera fácil métricas y rediseños aplicables a un LVDE dado. En el caso de métricas, proporcionan a lo sumo librerías que permiten su codificación [78] a mano desde cero, lo cual no es de gran ayuda teniendo en cuenta la gran variedad de métricas y de LVDEs a los que se pueden aplicar. Además, la codificación es propensa a la introducción de errores, e implica dominar el API del lenguaje de programación utilizado en la herramienta. Respecto a los rediseños, existen algunas herramientas de meta-modelado que integran lenguajes de transformación de modelos que se pueden usar con este objetivo. Sin embargo, los rediseños así especificados no se integran de manera automática en el entorno de modelado generado, sino que éste se debe modificar a posteriori para incluirlos [51, 118]. En otros casos utilizan herramientas de transformación de modelos que ni siquiera están integradas en el entorno generado, lo que requiere o bien trabajar con

los dos entornos (de modelado y de transformación) en paralelo, o bien definir mecanismos de traducción entre los dos, o hacer que ambos utilicen el mismo formato de representación de modelos (por ejemplo XMI). Así pues, de todo lo expuesto podemos deducir que se requieren nuevas herramientas de meta-modelado que faciliten al diseñador del entorno visual la tarea de definir métricas y rediseños para sus LVDEs, proporcionando para ello mecanismos de alto nivel, preferiblemente gráficos, que eliminen la necesidad de codificación. Sería igualmente interesante que esas herramientas permitieran establecer relaciones entre métricas y rediseños, de tal modo que ciertos valores extremos en las primeras pudieran forzar o simplemente sugerir la aplicación de un conjunto de los segundos. Esto es algo que no está presente en ninguna de las herramientas de meta-modelado existentes. Finalmente, los entornos de modelado generados deberían integrar automáticamente dispositivos para ejecutar las métricas y rediseños especificados.

1.2. Objetivos

El principal objetivo de esta tesis es la formalización de diversos aspectos avanzados en la generación de entornos para LVDEs definidos mediante meta-modelado. En concreto, pretende definir un conjunto de técnicas que permitan, a partir de una definición intuitiva, visual y formal, la generación de entornos para LVDEs con mecanismos integrados que ayuden a incrementar la calidad de los modelos construidos en tales entornos. Esto incluye mecanismos para garantizar la consistencia estática y dinámica entre modelos, para el análisis y la verificación de modelos (donde las preguntas y respuestas deben poder darse en términos del LVDE), y para medir y mejorar las características de calidad.

Para conseguir este propósito se plantean los siguientes objetivos generales:

(O1) Generación automática de entornos para LVDEs basados en meta-modelado, que permitan la definición de lenguajes multi-vista. Este objetivo se llevará a cabo mediante la consecución de los siguientes objetivos parciales:

- (O1.1) Formalización de lenguajes visuales multi-vista.
- (O1.2) Definición de un mecanismo para garantizar la consistencia estática entre vistas. El mecanismo debe ser flexible para permitir diversos paradigmas de comportamiento.
- (O1.3) Definición de un mecanismo para garantizar la consistencia dinámica entre vistas.
- (O1.4) Definición de un mecanismo para la verificación de propiedades semánticas. El mecanismo debe proporcionar técnicas flexibles para definir la forma de expresar los resultados, incluyendo en términos del LVDE.

(O1.5) Definición de un lenguaje visual de consultas sobre modelos, donde los resultados sean a su vez modelos que se actualicen para reflejar cambios del modelo consultado.

(O1.6) Construcción de una herramienta de soporte del marco.

(O2) Generación automática de entornos para LVDEs basados en meta-modelado, que integren mecanismos para medir y mejorar la calidad de los modelos. Este objetivo se llevará a cabo mediante la consecución de los siguientes objetivos parciales:

(O2.1) Definición de un catálogo de métricas estáticas independientes del LVDE.

(O2.2) Definición de un catálogo de rediseños independientes del LVDE.

(O2.3) Definición de un mecanismo para personalizar y configurar las métricas y rediseños del catálogo para un LVDE en concreto.

(O2.4) Definición de un mecanismo para la detección de las partes candidatas a ser rediseñadas en un modelo.

(O2.5) Construcción de una herramienta de soporte del marco.

Todo lo aquí expuesto será desarrollado y justificado en los diferentes capítulos que conforman esta tesis.

1.3. Método de trabajo

La presente tesis se ha elaborado según la metodología presentada en [166], cuyos pasos principales son: determinación del problema, formulación de la hipótesis, validación de la hipótesis y análisis de resultados, los cuales se detallan a continuación.

Determinación del problema *(capítulos 2 y 3).* Este trabajo surgió tras un estudio de las herramientas y enfoques existentes para la generación de entornos de modelado para LVDEs. Como ya se ha explicado, se detectó que en muchos casos los entornos generados eran meros editores de diagramas. Esto es insuficiente para el modelado de sistemas en general, y especialmente en paradigmas de desarrollo dirigido por modelos, donde se requieren entornos que ayuden activamente a la obtención de modelos con un alto nivel de calidad, por ejemplo garantizando su consistencia o proporcionando mecanismos de análisis y medición. También se detectó que estas herramientas adolecían con frecuencia de fundamentos sólidos en métodos formales. Por último, en muchos casos especificar el entorno requería un esfuerzo importante de codificación en algún lenguaje de programación textual, haciendo necesario conocer el API

apropiado según la herramienta utilizada. Sin embargo, como defienden los paradigmas que usan estas herramientas, usar notaciones visuales ayudaría a minimizar el esfuerzo de realizar tales especificaciones.

Formulación de la hipótesis *(capítulo 4)*. Así pues, la hipótesis de la que se partió en esta tesis es que es posible especificar y generar herramientas de modelado con capacidad para gestionar la calidad de los modelos, utilizando para ello mecanismos de alto nivel y formales. Además, la tesis seguiría la filosofía de que "todo es un modelo", en el sentido de que la funcionalidad que se quisiera generar para una herramienta debía especificarse mediante un modelo expresado en el LVDE más apropiado.

Para verificar la corrección de la propuesta se decidió extender la herramienta de meta-modelado AToM^3 [51] para permitir la definición de LVDEs según el marco propuesto, y generar los correspondientes entornos visuales.

Validación de la hipótesis *(capítulo 5)*. Partiendo de la hipótesis expuesta se definió el marco que recoge este documento. El marco se validó empíricamente mediante la especificación y generación de entornos visuales para LVDEs multi-vista en distintos áreas de aplicación. Para ello se usó la extensión de AToM^3 realizada en este trabajo de tesis. Dichos entornos se realizaron dentro del ámbito de dos proyectos de investigación. El primero de ellos fue "MDD-SIGE: Desarrollo basado en modelos de sistemas de información en Web para la gestión de emergencias" con referencia CCG06-UC3M/TIC-0787, y financiado por la Comunidad de Madrid y la Universidad Carlos III de Madrid. El segundo fue "ModuWeb: Desarrollo basado en modelos de sistemas Web de tele-educación", con referencia TIN2006-09678, y financiado por el Ministerio de Educación y Ciencia, Dir. Gral. de Investigación.

Análisis de resultados *(capítulo 6)*. Dichos entornos se definieron en colaboración con expertos en los dominios respectivos, validando de este modo la generalidad del enfoque mediante su aplicación a distintos campos. Además, los resultados obtenidos se han publicado no sólo en foros de meta-modelado y transformación de grafos (técnicas fundamentales en la solución aportada), sino también en otros orientados al dominio de aplicación de los LVDEs utilizados (véase siguiente sección). Esto demuestra la utilidad del enfoque en situaciones reales.

1.4. Publicaciones obtenidas

El presente trabajo ha dado lugar a las siguiente publicaciones:

- Revistas internacionales:

1. *"Generación de entornos de modelado avanzados mediante técnicas de transformación de grafos"*. 2006. Esther Guerra, Paloma Díaz y Juan de Lara. Revista IEEE América Latina, vol. 4(2), edición especial JISBD'2005. Versión extendida y traducida del artículo *"Supporting the automatic generation of advanced modelling environments with graph transformation techniques"*, pp.: 67-74 de las actas de las X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'05).
2. *"Event-driven grammars: Relating abstract and concrete levels of visual languages"*. 2007. Esther Guerra y Juan de Lara. Software and Systems Modeling (Springer), edición especial ICGT'2004. pp.: 317-347. Versión extendida del artículo *"Event-driven grammars: Towards the integration of meta-modelling and graph transformation"*, LNCS 3265, pp.: 54-69, presentado en 2^{nd} International Conference on Graph Transformation (ICGT'04).
3. *"Visual specification of measurements and redesigns for domain specific visual languages"*. 2007. Esther Guerra, Juan de Lara y Paloma Díaz. Próxima publicación en Journal of Visual Languages and Computing (Elsevier Science). 38 páginas.

- Conferencias internacionales:

1. *"Meta-modelling, graph transformation and model checking for the analysis of hybrid systems"*. 2003. Juan de Lara, Esther Guerra y Hans Vangheluwe. LNCS 3062, Springer. pp.: 292-298. Presentado en 2^{nd} Applications of Graph Transformation with Industrial Relevance (AGTIVE'03).
2. *"A formal approach to the generation of visual language environments supporting multiple views"*. 2005. Esther Guerra, Paloma Díaz y Juan de Lara. Actas del 2005 IEEE International Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05). pp.: 284-286.
3. *"A multi-view component modelling language for systems design: Checking consistency and timing constraints"*. 2005. Juan de Lara, Esther Guerra y Hans Vangheluwe. Actas del 2005 Workshop on Visual Modeling for Software Intensive Systems (VMSIS'05). pp.: 27-34.
4. *"Model transformation by graph transformation: A comparative study"*. 2005. Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan de Lara, Laszlo Lengyel, Tihamer Levendovszky, Ulrike Prange, Daniel Varro y Szilvia Varro-Gyapay. Presentado en Model Transformation in Practice (MTiP'05).
5. *"Model view management with triple graph transformation systems"*. 2006. Esther Guerra y Juan de Lara. LNCS 4178, Springer. pp.: 351-366. Presentado en 3^{rd} International Conference on Graph Transformation (ICGT'06).

6. *"Graph transformation vs OCL for view definition"*. 2006. Esther Guerra y Juan de Lara. Presentado en 1^{st} International Workshop on OCL and Applications (WAFOCA'06).
7. *"Visual specification of metrics for domain specific visual languages"*. 2006. Esther Guerra, Paloma Díaz y Juan de Lara. Publicación prevista como ENTCS. Presentado en 5^{th} International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'06).
8. *"Model-driven development of digital libraries: Generating the user interface"*. 2006. Alessio Malizia, Esther Guerra y Juan de Lara. Presentado en 2^{nd} International Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI'06). Actas publicadas como Vol. 214 de CEUR.
9. *"Model-driven development of digital libraries: Validation, analysis and code generation"*. 2007. Esther Guerra, Juan de Lara y Alessio Malizia. Actas del 4^{th} International Conference on Web Systems and Technologies (WEBIST'07). pp.: 35-42. Seleccionado como uno de los mejores artículos de la conferencia, está prevista la publicación de una versión extendida como Lecture Notes in Business Information Processing, Springer.
10. *"A transformation-driven approach to the verification of security policies in web designs"*. 2007. Esther Guerra, Daniel Sanz, Paloma Díaz e Ignacio Aedo. LNCS 4607, Springer. pp.: 269-284. Presentado en 7^{th} International Conference on Web Engineering (ICWE'2007).

- Conferencias nacionales:

1. *"A framework for the verification of UML models. Examples using Petri Nets"*. 2003. Esther Guerra y Juan de Lara. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'03). pp.: 325-334.

- Capítulos de libro:

1. "Model-based development: Meta-modelling, transformation and verification". 2005. Juan de Lara, Esther Guerra y Hans Vangheluwe. Capítulo del libro *"Management of the object-oriented development process"*, Idea Group Publishers. Editado por Liping Liu y Boris Roussev.
2. "Meta-modelling and graph transformation for the definition of multi-view visual languages". 2007. Esther Guerra y Juan de Lara. Capítulo del libro *"Visual languages for interactive computing: Definitions and formalizations"*, Idea Group Publishers. Editado por Fernando Ferri.

3. "Integrating measurements and redesigns in the definition of domain specific visual languages". Publicación prevista en 2008. Esther Guerra, Juan de Lara y Paloma Díaz. Capítulo del libro *Model-driven software development: Integrating quality assurance*, Idea Group Publishers. Editado por Jörg Rech y Christian Bunse.

- Informes técnicos:

1. *"Attributed typed triple graph transformation with inheritance in the double pushout approach"*. 2006. Esther Guerra y Juan de Lara. Informe técnico UC3M-TR-CS-06-01 de la Universidad Carlos III de Madrid. 61 páginas.

1.5. Estructura del documento

A continuación se detalla la estructura de este documento, describiendo el objetivo fundamental de cada capítulo.

Capítulo 1. Introducción. Es el presente capítulo. Describe el contexto y la motivación que llevan al desarrollo de esta tesis. Incluye un resumen de objetivos, y la metodología de trabajo seguida en su elaboración.

Capítulo 2. Conceptos previos. Introduce brevemente algunos conceptos teóricos que conforman la base sobre la que se elabora esta tesis: lenguajes visuales de dominio específico, meta-modelado, transformación de modelos y calidad del software.

Capítulo 3. Estado del arte. Presenta un estudio de los principales enfoques existentes para conceptos clave en esta tesis, como son: mecanismos de consistencia en lenguajes visuales (multi-vista), herramientas para la generación de entornos visuales para LVDEs, lenguajes de transformación de modelos, herramientas CASE con capacidades de análisis, verificación y gestión de calidad de modelos, y herramientas para desarrollo dirigido por modelos.

Capítulo 4. Marco para la especificación, análisis y generación de entornos para lenguajes visuales de dominio específico. Este capítulo presenta la propuesta realizada para generar entornos visuales para LVDEs que integren mecanismos de control de calidad de los modelos construidos en tales entornos. Esto incluye consistencia entre modelos, análisis, consultas, medición de las características de calidad y mejora de las mismas. También describe la construcción de un prototipo que permite la generación de entornos visuales con tales funcionalidades a partir de descripciones de alto nivel.

Capítulo 5. Evaluación. Recoge la evaluación empírica del marco propuesto mediante la construcción de entornos para distintos LVDEs en el área de los sistemas web y en el de las bibliotecas digitales. También muestra la construcción de un entorno para un subconjunto del lenguaje de propósito general UML. La generación de los entornos se realiza con el prototipo desarrollado.

Capítulo 6. Conclusiones. Analiza el grado de cumplimiento de los objetivos presentados en el capítulo 1, enumera las aportaciones realizadas, y resume las conclusiones obtenidas tras la elaboración de la tesis. El capítulo finaliza indicando líneas futuras de investigación abiertas a partir del presente trabajo.

Capítulo 2

Conceptos previos

En este capítulo se incluye una introducción a distintos conceptos y técnicas clave para la elaboración de la presente tesis, ya sea en su planteamiento o en su solución.

La sección 2.1 comienza presentando el meta-modelado. Esta técnica se usará para alcanzar algunos de los objetivos marcados en esta tesis, sirviendo como base para especificar LVDEs y generar entornos visuales para ellos.

Ya que a menudo tales entornos suelen incorporar lenguajes para la manipulación de modelos, en la sección 2.2 se explica qué es la transformación de modelos, qué tipos hay y cuándo se utilizan. Se profundizará en la transformación de grafos como mecanismo formal, gráfico y de alto nivel para la manipulación de modelos, ya que junto al meta-modelado constituye la base teórica del presente trabajo.

A continuación, la sección 2.3 describe qué son los LVDEs, cómo se definen y las ventajas derivadas de su uso. También se discuten técnicas habituales para la generación de entornos que los soporten, básicamente el meta-modelado y la transformación de grafos.

Finalmente, la sección 2.4 estudia algunas técnicas habituales para controlar la calidad del producto software durante las etapas iniciales de su desarrollo, esto es, durante su análisis y diseño. El término "calidad" es amplio e incluye estándares, procedimientos, calidad de procesos, etc. Aquí sólo se presentarán técnicas que permiten comprobar, medir o mejorar la calidad de los modelos de diseño que representan un sistema, como por ejemplo métodos formales para su especificación y verificación, métricas para cuantificar características de calidad, patrones de diseño como mecanismo de reutilización del conocimiento en un dominio, y rediseños orientados a la mejora de los atributos de calidad. Los entornos de modelado deberían integrar tales técnicas para ayudar a los desarrolladores a la obtención de sistemas de calidad.

2.1. Meta-modelado

El meta-modelado [12] es un enfoque usado para describir y generar entornos para LVDEs. Esta técnica consiste en construir un modelo (llamado meta-modelo) que describe el conjunto de todos los modelos válidos del lenguaje. El meta-modelo se suele especificar usando notaciones visuales como diagramas de clases o entidad-relación, más restricciones adicionales expresadas en un lenguaje de restricciones que suele ser textual como OCL [189]. Por ejemplo, la figura 2.1 muestra una arquitectura de meta-modelado para el lenguaje "Autómatas Finitos". A la izquierda, el meta-modelo del lenguaje se ha definido mediante un diagrama de clases que contiene la definición de los elementos, relaciones y cardinalidades del lenguaje. El meta-modelo incluye una restricción OCL que especifica que el nombre de un estado debe ser único. A la derecha se muestra un modelo válido del lenguaje, esto es, un modelo "conforme" al meta-modelo dado.

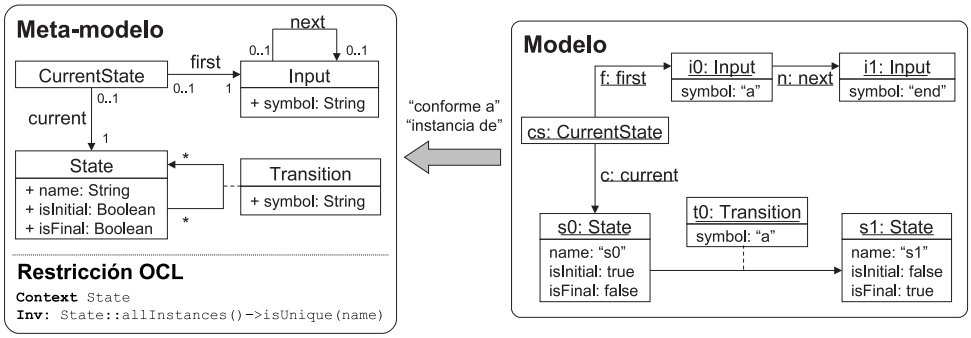


Figura 2.1: *Meta-modelo del lenguaje "Autómatas Finitos", y modelo conforme al meta-modelo*

Las arquitecturas basadas en meta-modelado se organizan en niveles. Cada nivel contiene la definición de los modelos que se encuentran en el nivel inmediatamente inferior, y éstos deben ser conformes a la descripción dada por el nivel superior. Aunque la arquitectura que muestra la figura 2.1 contiene sólo dos niveles (meta-modelo y modelo), la especificación de UML 2.0 [182] define 4 niveles que se muestran en la figura 2.2. El nivel más alto ($M3$) contiene los meta-meta-modelos que definen el lenguaje usado para especificar meta-modelos. Este nivel incluye mecanismos para definir entidades (meta-clases), datos (meta-atributos), operaciones (meta-operaciones) y relaciones (meta-relaciones). En la meta-arquitectura de UML esos elementos se organizan como el meta-meta-modelo MOF (*Meta-Object Facility*), similar a un subconjunto de los diagramas de clases UML. El nivel $M2$ contiene los meta-modelos que definen lenguajes de modelado. En este nivel se encuentran, por ejemplo, la descripción de los autómatas finitos, de los diagramas entidad-relación o de los diagramas UML. Los modelos conformes a alguno de los meta-modelos del nivel $M2$ pertenecen al nivel $M1$. Por ejemplo, la figura muestra diversos modelos entidad-relación escritos usando el meta-modelo de diagramas entidad-relación definido en el nivel superior. Finalmente, $M0$ contiene los datos resultantes de la ejecución de los modelos del nivel $M1$.

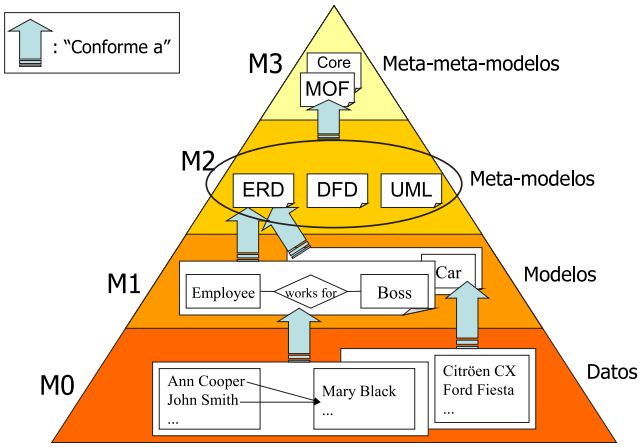


Figura 2.2: *Arquitectura de meta-modelado en 4 niveles*

Por otro lado, para facilitar la especificación de sistemas software complejos es habitual usar una notación distinta para capturar cada una de sus vistas o aspectos. A este tipo de lenguajes de modelado que incluyen un conjunto de notaciones diferentes se les denomina lenguajes multi-vista, entre los cuales UML2.0 es un ejemplo destacado. Cada notación individual está definida según su propio meta-modelo. Sin embargo, todas esas definiciones se basan en un meta-modelo único. Si los meta-modelos de dos notaciones solapan en el meta-modelo único, eso significa que existe una dependencia entre ellos ya que definen el mismo concepto desde diferentes puntos de vista. Estas intersecciones definen restricciones de consistencia que deben cumplir los modelos del nivel inferior.

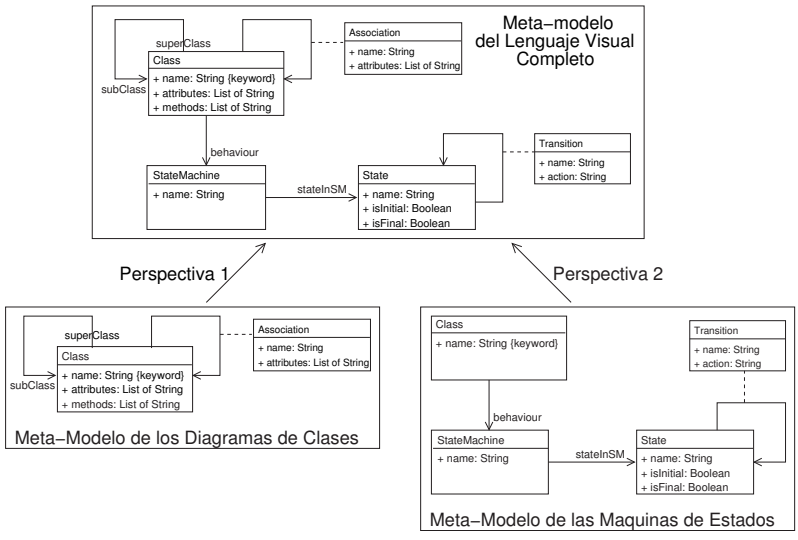


Figura 2.3: *Definición de un subconjunto de UML*

Por ejemplo, la figura 2.3 muestra la definición de un pequeño subconjunto de UML me-

diante el meta-modelo de la parte superior. Este meta-modelo contiene clases, asociaciones binarias, relaciones de herencia entre clases, máquinas de estados, estados y transiciones. Como puede verse, el meta-modelo mezcla aspectos dinámicos y estructurales de la definición de un sistema. Para facilitar la especificación de sistemas, en la parte inferior se definen dos lenguajes (o tipos de diagrama) sobre el meta-modelo completo: uno para especificar la estructura estática (diagramas de clases) y otro para especificar el comportamiento (máquinas de estados). El primero contiene clases, asociaciones y relaciones de herencia, mientras que el segundo sólo contiene máquinas de estados, estados, transiciones y las clases a las que se asocian las máquinas de estados. Puede verse que el concepto clase está presente en los dos lenguajes, y que de hecho en el segundo caso sólo define un atributo nombre. Sin embargo ambos se refieren al mismo concepto en el meta-modelo único, y por tanto existe una dependencia indirecta entre los dos sub-lenguajes.

2.1.1. Herramientas de meta-modelado

Una de las principales aplicaciones del meta-modelado es la generación de herramientas de modelado (o herramientas CASE) para un formalismo dado, a partir de la descripción de su sintaxis mediante un meta-modelo [50, 118]. El meta-modelado permite definir rápida y fácilmente entornos visuales para la manipulación de lenguajes visuales que se ajustan tanto a dominios específicos como a las necesidades del desarrollador. Al trabajar a mayores niveles de abstracción se consigue reducir el desarrollo de código al mínimo, aumentando la productividad y calidad del producto, y minimizando los costes de codificación, pruebas y mantenimiento.

Las herramientas de meta-modelado (también llamadas metaCASE) se basan en una arquitectura de tres capas que se corresponden con los niveles de meta-modelado $M3$-$M2$-$M1$. La capa más alta incluye el lenguaje utilizado para definir los meta-modelos, el cual suele ser gráfico (por ejemplo diagramas de clases o entidad-relación) y está normalmente fijo. La capa intermedia contiene el meta-modelo del formalismo, que es especificado por el usuario de la herramienta. Finalmente, en la capa inferior están las herramientas de modelado generadas automáticamente a partir de los meta-modelos, las cuales permiten la manipulación de modelos que son instancias del formalismo descrito.

Respecto a los entornos generados mediante meta-modelado, existen tres opciones (que pueden usarse en combinación) para dotarles de funcionalidad. En el primer caso, el entorno generado está formado por un núcleo central predefinido que se completa con código específico generado a partir del meta-modelo del formalismo. Este enfoque es adecuado si el dominio de la aplicación se conoce bien, pero no permite añadir funcionalidades adicionales específicas del entorno generado. Para evitar este problema, hay casos en que como segunda opción se permite desarrollar funciones en el lenguaje de implementación del entorno generado, funciones que posteriormente se integrarán en el mismo. La desventaja es que

estos lenguajes son de bajo nivel (en comparación con la notación usada para describir el meta-modelo) y requiere conocer el API del lenguaje de implementación. Finalmente, la tercera opción es incluir información sobre la funcionalidad adicional del entorno generado en la capa de meta-modelado, usando para ello una notación de alto nivel. Elevar el nivel de abstracción lleva a un menor tiempo de desarrollo de las nuevas funcionalidades, la curva de aprendizaje es menor que la de notaciones de bajo nivel, y la calidad del entorno resultante tiende a incrementarse.

2.2. Transformación de modelos

La transformación de modelos consiste en transformar un modelo origen conforme a un meta-modelo, en un modelo destino conforme a otro meta-modelo (que puede ser distinto o el mismo). Este tipo de transformaciones es uno de los pilares del Desarrollo de Software Dirigido por Modelos (DSDM), donde se utilizan para transformar modelos independientes de la plataforma en modelos dependientes de la plataforma [131, 188], simulación, optimización y análisis del modelo destino. En particular, la transformación de modelos está teniendo un gran auge a partir de propuestas específicas para el DSDM como la promovida por la OMG con su arquitectura dirigida por modelos (MDA, *Model-Driven Architecture*) [131, 188] y su lenguaje para consultas, vistas y transformaciones QVT [155].

Las transformaciones de modelos se pueden clasificar según dos características ortogonales: los meta-modelos fuente y destino de la transformación y el nivel de abstracción. Si nos fijamos en la primera característica podemos tener transformaciones que son:

- *Inter-formalismo o exógenas* [134], si los meta-modelos origen y destino son distintos. Este tipo de transformaciones se puede utilizar con propósitos de análisis, de tal modo que la transformación se realiza a un dominio semántico que proporciona herramientas para el estudio de propiedades sobre el modelo original. En este caso, la transformación debe preservar las características bajo investigación. Otros campos de uso de este tipo de transformaciones son la migración (por ejemplo a versiones posteriores de un lenguaje), o la transformación de modelos independientes de la plataforma a dependientes de la plataforma. Finalmente, la simulación también se podría considerar una transformación exógena iterativa desde un formalismo fuente a un formalismo "trazas", donde una traza es una secuencia ordenada de eventos que contienen información sobre el estado de las variables del sistema simulado, así como el tiempo de simulación.

- *Intra-formalismo o endógenas* [134], si los meta-modelos origen y destino son el mismo. Estas transformaciones se utilizan para optimización, simplificación, rediseño de modelos, animación, etc.

Dependiendo del nivel de abstracción de los modelos fuente y destino podemos tener transformaciones que son:

- *Horizontales*, si transforman el modelo a otro formalismo con el mismo nivel de abstracción. Por ejemplo, el rediseño de modelos (que es intra-formalismo) o la migración (que es inter-formalismo) pertenecen a esta categoría.

- *Verticales*, si el modelo destino tiene un nivel de abstracción distinto que el origen. La generación de código se considera en muchas ocasiones una transformación vertical desde una notación con un mayor nivel de abstracción (el modelo) a uno de

menor abstracción (el código). En la dirección opuesta podemos citar como ejemplo la ingeniería inversa, que es una transformación vertical desde un nivel de abstracción menor (el código) a uno mayor (el diseño).

Existen numerosos lenguajes de transformación de modelos tanto visuales [6, 142, 159, 162] como textuales [14, 44], y definidos según una base formal [17, 159, 162] o semi-formal [6, 14, 142]. En cualquier caso, existen ciertas propiedades deseables en una transformación de modelos que sólo son verificables bajo la perspectiva de un lenguaje formal. Las propiedades de corrección más típicas de una transformación son las siguientes:

- *terminación*: la transformación termina en una cantidad finita de tiempo.
- *confluencia*: partiendo de un modelo origen, el resultado de la transformación es único.
- *consistencia sintáctica*: el modelo resultante de la transformación es conforme al meta-modelo destino.
- *consistencia semántica*: la transformación preserva las propiedades semánticas de interés.

Esta tesis persigue la utilización de técnicas formales y visuales para la especificación y generación de herramientas de modelado para LVDEs. En esos entornos, los lenguajes de transformación de modelos se pueden utilizar para la manipulación de modelos. Dentro de ese contexto, la transformación de grafos [159] resulta ser un lenguaje de transformación de modelos que presenta ambas características: ser visual y formal (además de declarativo). Por esa razón, la solución propuesta en esta tesis tiene como base la transformación de grafos, cuyos principales conceptos se introducen a continuación.

2.2.1. Transformación de grafos

Ya que los modelos, meta-modelos y meta-meta-modelos se pueden representar como grafos con tipos y atributos, es posible utilizar transformación de grafos para realizar transformación de modelos de manera gráfica, intuitiva, declarativa y formal [159]. Existen ejemplos de su aplicación en áreas muy diversas, como la transformación de modelos [64], el modelado de entornos para lenguajes visuales [138], la simulación visual [52], la formalización de *refactorings* [104, 133], la biología [158], la seguridad [160], los servicios web [39], etc.

Las gramáticas de grafos son la extensión natural de las gramáticas de Chomsky (sobre cadenas) al dominio de los grafos. Están formadas por reglas de producción declarativas que contienen grafos en sus partes izquierda (L) y derecha (R). La aplicación de una regla

a un grafo anfitrión consiste en encontrar un morfismo válido desde su parte izquierda al grafo, y sustituirlo por su parte derecha. Las reglas de una gramática se aplican de manera no determinista en el grafo anfitrión hasta que no es posible aplicar ninguna más. La semántica de una gramática de grafos son todos los grafos posibles que pueden resultar de la aplicación de las reglas sobre un grafo inicial.

Los sistemas de transformación de grafos pueden mostrar dos clases de no-determinismo. Primero, la parte izquierda de la regla puede encontrarse en distintas partes del grafo anfitrión. En ese caso, o bien la regla se aplica en paralelo en todas las partes si éstas son disjuntas, o bien se aplica en sólo una de ellas, la cual puede elegirse aleatoriamente o ser elegida por el usuario. La segunda fuente de no-determinismo es que dos o más reglas pueden ser aplicables en un determinado momento. En principio, el orden en que se aplican las reglas de una gramática de grafos es arbitrario. Sin embargo, existen técnicas para controlar el orden de ejecución de las reglas tales como la utilización de lenguajes de control [102], el uso de prioridades en las reglas, y la división de reglas en capas.

La teoría sobre transformación de grafos se viene desarrollando desde la década de los 70. La forma de las reglas, el algoritmo de búsqueda empleado sobre el grafo, y el mecanismo de aplicación de las reglas, determinan los distintos enfoques teóricos existentes para la transformación de grafos. Brevemente, los principales enfoques son [159]:

- *Sistemas de reetiquetado de nodos*: las reglas en este tipo de sistemas no cambian la morfología del grafo anfitrión, sino sólo el etiquetado de sus vértices y/o relaciones. La aplicación de una regla modifica un subgrafo conexo de tamaño fijo en función del contexto.

- *Gramáticas de hipergrafos*: un hipergrafo es una generalización de un grafo dirigido con un conjunto finito de nodos y un conjunto finito de hiper-relaciones. Cada hiper-relación tiene una etiqueta y está conectada mediante vértices a distintos nodos. Las reglas de una gramática de este tipo contienen una hiper-relación en su parte izquierda que se sustituye por el hipergrafo que se especifica en la parte derecha (esto es, son libres de contexto).

- *Enfoque algebraico*: modela la aplicación de una regla en un grafo mediante construcciones *pushout* (PO). Un PO es una construcción categórica que, en el caso de grafos, se construye como la unión de dos grafos a través de un subgrafo común (véase apéndice A). Existen dos variantes: *single-* y *double-pushout* (SPO y DPO respectivamente) según se utilicen uno o dos POs en una derivación directa.

- *Enfoque basado en lógica*: en este enfoque las reglas y propiedades de los grafos se expresan en lógica monádica de segundo orden.

- *Teoría de 2-estructuras*: marco para la descomposición y transformación de sistemas matemáticos donde los objetos del sistema están relacionados mediante una o más relaciones binarias, como por ejemplo ocurre con los grafos.

- *Sistemas de reemplazo de grafos programados*: combinan los aspectos habituales de la transformación de grafos con estructuras de control imperativas que controlan la elección no-determinista en el orden de aplicación de las reglas.

Entre los enfoques mencionados, una de las formalizaciones más populares es el *Double Pushout* (DPO) [61], que utiliza teoría de categorías para modelar reglas y derivaciones. El siguiente apartado presenta de manera intuitiva dicho enfoque.

Enfoque algebraico *Double Pushout*

Una regla de reescritura o producción en DPO es una terna de componentes $p = L \xleftarrow{l} K \xrightarrow{r} R$. L y R se denominan parte izquierda y derecha de la regla, respectivamente, y l y r son morfismos normalmente inyectivos. La parte izquierda describe las precondiciones necesarias para aplicar la regla, mientras que la parte derecha representa postcondiciones. El grafo interfaz K contiene los elementos comunes en L y R, esto es, los elementos que la aplicación de la regla preserva. Intuitivamente, $L - K$ contiene los elementos que se eliminarán al aplicar la regla, mientras que $R - K$ contiene los que se crearán.

A grandes rasgos, una *transformación de grafos directa* con una producción p se realiza en 3 pasos. Primero, debe encontrarse un morfismo m desde la parte izquierda L de la regla al grafo anfitrión G. Después, todos los vértices de $L - K$ se borran de G, obteniendo el grafo intermedio D. Por último, D y $R - K$ se pegan a través de los elementos identificados por K para obtener el grafo resultante de la transformación. Una *transformación de grafos* es una secuencia de cero o más transformaciones de grafos directas.

Formalmente, una derivación directa para una producción $p = L \xleftarrow{l} K \xrightarrow{r} R$ dado un morfismo $m\colon L \to G$ se construye mediante dos POs, tal y como muestra el diagrama de la figura 2.4. Primero, el PO (1) elimina de G los elementos de $L - K$, obteniendo el grafo de contexto D. Después, el PO (2) añade al grafo de contexto los elementos de $R - K$, obteniendo el grafo H resultante y completando la derivación directa $G \Rightarrow H$ vía p y m. Nótese que, en realidad, en el primer paso lo que se calcula es el complemento PO. Para que dicho complemento exista es necesario que se cumplan dos condiciones adicionales, denominadas *"enlaces colgantes"* y *condición de identificación*. La primera condición especifica que si la aplicación de una regla no borra un enlace, entonces los nodos origen y destino del enlace también se tienen que preservar (de otro modo quedaría un enlace "colgando"). La condición de identificación especifica que si dos nodos o enlaces del grafo interfaz K se identifican con un único elemento del grafo anfitrión (mediante un morfismo no inyectivo),

entonces ambos elementos se tienen que borrar o preservar, ya que si uno se elimina y el otro se mantiene se produce un conflicto.

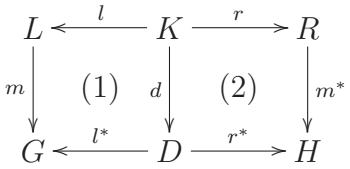


Figura 2.4: *Derivación directa en el enfoque* Double Pushout

Una secuencia $G_0 \Rightarrow G_1 \Rightarrow ... \Rightarrow G_n$ de derivaciones directas se llama *derivación*, y se denota como $G_0 \overset{*}{\Rightarrow} G_n$.

La figura 2.5 muestra la aplicación de una regla a un grafo anfitrión G. Tanto los componentes de la regla como el grafo se muestran utilizando una sintaxis concreta, donde algunos atributos se representan como decoraciones de los nodos y relaciones (por ejemplo, el valor asociado a una transición se dibuja como una decoración sobre la misma). La regla, que se muestra en la parte superior, especifica un paso en la simulación de un autómata finito. Su parte izquierda muestra dos estados etiquetados `1` y `3`, conectados a través de una transición con etiqueta `2`. El nodo `4` tiene una relación que apunta al estado actual y otra que apunta a una cola de entradas. Si el valor de la primera entrada es igual al valor de la transición (variable `x`), la regla consume la primera entrada de la cola y cambia el estado actual. A lo largo de este documento se usará la misma notación que en la figura, etiquetando sólo los elementos que la regla preserva. La figura muestra la aplicación de la regla a un grafo con 3 estados. La imagen de L en el grafo se muestra coloreada. En la derivación, la variable `x` de la regla toma el valor `b`, y el estado actual se mueve de $\mathtt{s}_0$ a $\mathtt{s}_1$. La derivación se realiza en dos pasos: primero borra los elementos que están en L y no en K, obteniendo el grafo D, y a continuación añade a D los elementos de R que no están en K, dando como resultado el grafo H. La regla puede volver a aplicarse en H si los estados `1` y `3` de L toman como valor el estado $\mathtt{s}_2$ del grafo mediante un morfismo no inyectivo.

Como puede observarse, el componente K de una regla se puede calcular como el conjunto de elementos que tienen las mismas etiquetas en las partes derecha e izquierda de la regla. Por ello, en lo que resta de documento se omitirá mostrar K en las reglas. También es bastante usual dibujar las reglas utilizando una notación compacta en la que los componentes L y R se muestran en un solo grafo. En ese caso, los elementos que la regla crea aparecen etiquetados como "new", y los elementos que borra aparecen etiquetados como "del". Por ejemplo, la figura 2.6 muestra la regla de la figura 2.5 en su forma compacta.

El contexto en que una regla es aplicable se puede restringir mediante el uso de condiciones de aplicación [61]. Una condición de aplicación está formada por un grafo premisa y un conjunto de grafos consecuencia. Dada una imagen de la parte izquierda de una regla en un grafo, si es posible encontrar la premisa en el grafo, entonces es necesario encontrar

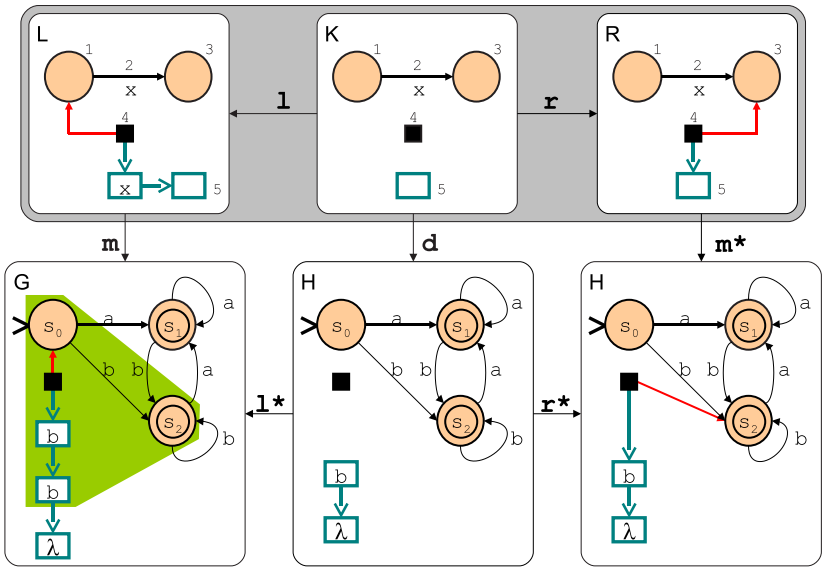


Figura 2.5: *Ejemplo de derivación directa*

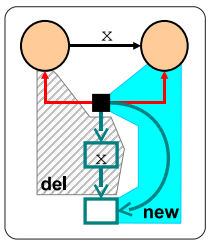


Figura 2.6: *Ejemplo de regla en notación compacta*

la imagen de alguna de las consecuencias para que la instancia de la parte izquierda sea válida (y la regla se pueda aplicar). Esto debe cumplirse para todas las condiciones de aplicación definidas para la regla. Hay dos condiciones de aplicación especiales. La primera se da cuando se especifica una premisa pero ninguna consecuencia, y se denomina Condición de Aplicación Negativa (NAC). En ese caso, encontrar una imagen de la premisa hace que la regla no se pueda aplicar. El segundo tipo se denomina Condición de Aplicación Positiva (PAC), y se da cuando el grafo premisa es isomorfo a la parte izquierda y además existe algún grafo consecuencia. En este caso, es necesario encontrar al menos una imagen de alguna de las consecuencias para que la regla se pueda aplicar.

De manera formal, una condición de aplicación está formada por un morfismo $x\colon L \to X$ de L al grafo premisa X, un conjunto de grafos consecuencia Y_i, y morfismos y_i desde X a cada Y_i: $c = \{L \xrightarrow{x} X, X \xrightarrow{y_i} Y_i\}$, tal y como muestra el diagrama de la figura 2.7. La condición de aplicación satisface el morfismo m si no existe un morfismo $n\colon X \to G$ tal que $n \circ x = m$. Si tal n existe, entonces la condición de aplicación satisface m sólo si existe

un morfismo $o\colon Y_i \to G$ tal que $o \circ y_j = n$ para alguno de los grafos consecuencia Y_i.

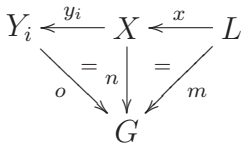


Figura 2.7: *Condición de aplicación de una regla*

La figura 2.8 muestra un ejemplo de derivación directa para una regla con NAC. El morfismo entre la LHS y la NAC de la regla se representa mediante los elementos etiquetados con los mismos números (en este caso sólo el nodo `1`). La regla crea un puntero al estado inicial, y la NAC controla que la regla se aplique sólo si no existe tal puntero. De este modo nos aseguramos de que la regla se aplica una sola vez al principio de la simulación del autómata. La figura muestra la aplicación de la regla a un grafo anfitrión G en el subgrafo que se muestra coloreado. Como no es posible encontrar una instancia de la NAC para esa imagen (es decir, $\mathtt{s}_0$ no tiene ningún puntero asociado) podemos aplicar la regla y obtener el grafo H. Aunque en H aún encontramos la misma imagen de la LHS de la regla, en este caso también es posible encontrar la NAC (es decir, $\mathtt{s}_0$ ya tiene un puntero asociado), y por tanto la regla no se puede aplicar.

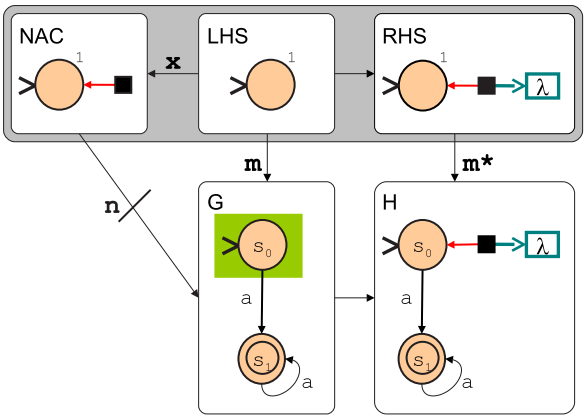


Figura 2.8: *Ejemplo de derivación directa mediante regla con condición de aplicación negativa*

Para incrementar la expresividad de las reglas es posible combinar la transformación de grafos con el concepto de herencia existente en meta-modelado, en las llamadas *reglas abstractas* [45]. En esos casos, los elementos que pertenecen a la parte izquierda de la regla pueden tener como imagen cualquier objeto de las clases hijas de su clasificador. De ese modo, una sola regla equivale al conjunto de reglas que resultan de sustituir sus elementos por objetos de las clases en la jerarquía de herencia. La figura 2.9 muestra en la parte superior un ejemplo de regla abstracta que elimina un permiso asignado a un

sujeto si dicho sujeto forma parte de otro (relación *composicion*) que también lo define. En la regla, los nodos de tipo sujeto pueden tener como imagen un equipo o un rol (clases hija del sujeto según el meta-modelo de la izquierda), y el componente función puede ser una función sencilla o compuesta. En la derivación que se muestra, el sujeto 1 tiene como imagen un equipo en el grafo anfitrión, el sujeto 2 tiene un rol, y el componente función asume una función sencilla.

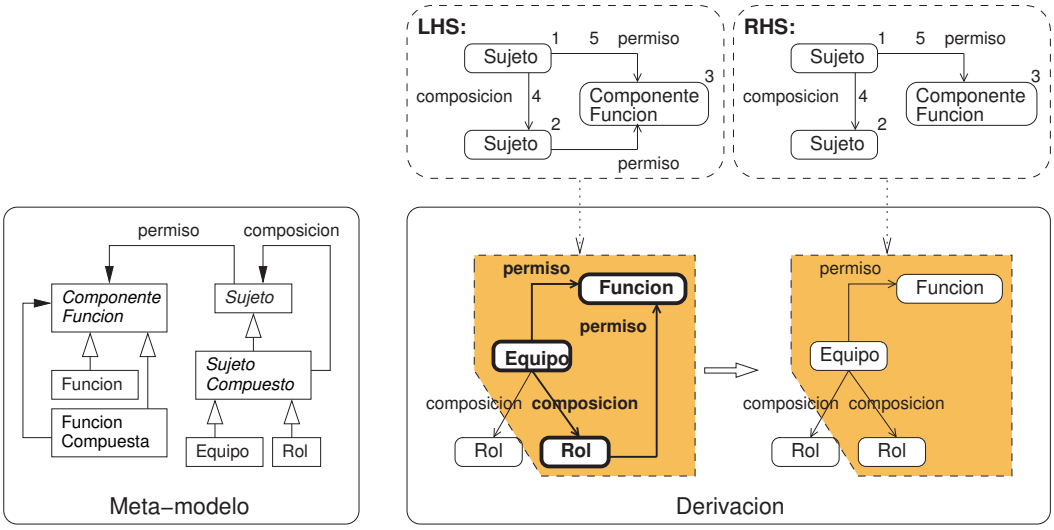


Figura 2.9: *Ejemplo de regla abstracta y derivación*

Otra variante de la transformación de grafos es la transformación de grafos paralela [177], que permite instanciar un número arbitrario de veces ciertas partes de una regla, y aplicarla en todas las instancias de manera síncrona. Una *regla paralela* está formada por un conjunto de reglas elementales, un conjunto de subreglas, y una relación de integración entre las subreglas y las reglas elementales (esto es, entre sus componentes L, K y R). La regla elemental se puede instanciar un número arbitrario de veces en el grafo anfitrión, siempre que las instancias se solapen para todas las subreglas.

Recientemente, los resultados obtenidos para el enfoque DPO se han generalizado para que no sólo grafos, sino cualquier objeto de una categoría HLR adhesiva [110], pueda ser reescrito utilizando transformación de grafos [61]. Ejemplos de este tipo de categoría son los grafos con atributos, los grafos con tipos, los hipergrafos y las redes de Petri. El interés es que, en consecuencia, cualquiera de estas estructuras se puede manipular con la misma teoría DPO.

Técnicas de análisis en el enfoque *Double Pushout*

La naturaleza formal de las gramáticas de grafos permite analizar propiedades de las transformaciones. Al comienzo de esta sección se enumeraban algunas características desea-

bles en una transformación de modelos, como su terminación, confluencia o consistencia. A continuación se presentan algunos de los principales resultados en el enfoque DPO para el estudio de este tipo de propiedades.

Una propiedad interesante en una transformación de modelos es la *terminación* de su ejecución, lo cual es indecidible en general [151]. Sin embargo, seguir ciertos criterios al construir una gramática de grafos permite demostrar su terminación [59, 61]. Para ello, el conjunto de símbolos y reglas se divide en capas, y se restringe la capa de símbolos que las reglas de una capa pueden crear y borrar.

También resulta interesante saber si una transformación es *confluente*, es decir, si empezando desde un modelo inicial se obtiene un único modelo resultante [61]. En ese caso no importa que haya diversos caminos en la ejecución de la transformación, siempre que todos lleven al mismo resultado. Si una transformación es confluente y termina se dice que tiene comportamiento funcional. Para estudiar la confluencia se puede usar análisis de pares críticos [92]. Sean dos reglas p_1 y p_2 con morfismos m_1 y m_2 de las reglas a un grafo G. Se dice que la pareja de transformaciones $p_1(m_1) : G \Rightarrow H_1$ y $p_2(m_2) : G \Rightarrow H_2$ son un *par crítico* si están en conflicto (esto es, p_1 puede deshabilitar la aplicación de p_2 o viceversa), siendo G mínimo. Hay tres razones por las que dos reglas pueden estar en conflicto: (i) una de ellas borra un objeto que está incluido en el morfismo donde se aplica la otra; (ii) una crea objetos que aparecen en la NAC de la otra, y por tanto pueden deshabilitarla [112]; (iii) una cambia atributos que pertenecen a la imagen de la parte izquierda de la otra regla. El análisis de pares críticos busca todas las aplicaciones de reglas para grafos críticos mínimos en las que las reglas se pueden aplicar y generar un conflicto. Una gramática es confluente si no tiene pares críticos, o si para cada par crítico se puede demostrar que o bien no se puede producir en un caso real (por ejemplo por restricciones del lenguaje), o bien a partir de los dos estados que se obtienen sólo es posible llegar a un mismo estado.

En el caso de transformaciones inter-formalismo se necesita garantizar la corrección sintáctica y semántica del modelo destino. Eso significa que el modelo destino debe ser conforme al meta-modelo destino, y que la transformación debe preservar las propiedades semánticas de interés. En [49] se presenta una técnica para garantizar la consistencia sintáctica del modelo destino que se basa en definir un meta-modelo intermedio para los modelos que se generan durante la transformación. Este meta-modelo intermedio incluye los meta-modelos origen y destino y otros elementos auxiliares. De este modo se puede validar el comportamiento funcional del proceso de transformación, y se facilita la verificación de la consistencia y equivalencia de comportamiento.

En el campo de las transformaciones que implementan simuladores para expresar el comportamiento de un modelo, es interesante analizar la *concurrencia* de las reglas [16]. Ésta puede estudiarse desde los puntos de vista secuencial y explícito. En el primer caso, el concepto de independencia secuencial establece cuándo dos reglas se pueden aplicar en distinto orden y obtener el mismo resultado. En cambio, el enfoque explícito busca realizar

varias acciones simultáneamente. Para ello, el teorema del paralelismo establece cuándo dos reglas se pueden componer en una regla paralela que se aplica en un solo paso, y cuándo se deben aplicar de manera secuencial.

Por último, también se pueden usar *restricciones globales* para modelar con grafos cualquier estado prohibido que no queremos que ocurra en un sistema. Al aplicar una transformación, las restricciones se transforman en condiciones de aplicación para cada una de las reglas de la transformación [93]. Esas condiciones se encargan de que las reglas no se apliquen si van a dejar al sistema en un estado no deseado.

Gramáticas de grafos triples

Las Gramáticas de Grafos Triples (TGGs) fueron inventadas por Andy Schürr [162] como un mecanismo para especificar traductores de estructuras de datos, mantener la consistencia, y propagar pequeños cambios de una estructura de datos a otra mediante actualizaciones incrementales. Permiten definir reglas declarativas de alto nivel para la creación sincronizada de elementos en dos grafos (origen y destino) relacionados a través de un grafo correspondencia. A partir de esas reglas de creación, y utilizando una serie de algoritmos, se genera un conjunto de reglas operacionales que permiten realizar la transformación en ambas direcciones (a partir de un grafo origen generan uno destino y viceversa) o crear el grafo correspondencia dados un grafo origen y otro destino existentes. Los componentes del grafo triple son grafos etiquetados atribuidos, donde los nodos del grafo correspondencia tienen morfismos (*mappings*) a los nodos de los grafos origen y destino. Además, las reglas de una TGG son monotónicas, eso es, permiten la creación pero no el borrado de elementos.

Por ejemplo, la figura 2.10 muestra a la izquierda una regla de TGG que especifica de manera declarativa la relación entre dos elementos de tipo *Tabla* y *Clase*. La regla contiene un grafo triple cuyos componentes se muestran separados con líneas verticales discontinuas (esto es, la tabla corresponde al grafo origen, la clase al destino, y su relación al grafo correspondencia). A partir de esa regla se generan las reglas triples operacionales de la derecha que permiten crear una clase si existe una tabla (primera regla), crear una tabla si existe una clase (segunda regla), o crear la relación de correspondencia entre una clase y una tabla existentes (tercera regla).

Las TGGs tienen potencial para ser muy útiles en transformaciones inter-formalismo [108], proporcionando una base formal para el análisis de las transformaciones. Utilizadas en ese contexto, los grafos origen y destino corresponderían a los modelos origen y destino de la transformación, y el proceso de transformación crearía en el grafo correspondencia relaciones entre los elementos de ambos. Sin embargo, algunas características de su formalización [162] imponen restricciones a la hora de usarlas para transformaciones inter-formalismo de carácter general. Por ejemplo, resultaría útil tener relaciones más flexibles entre el grafo

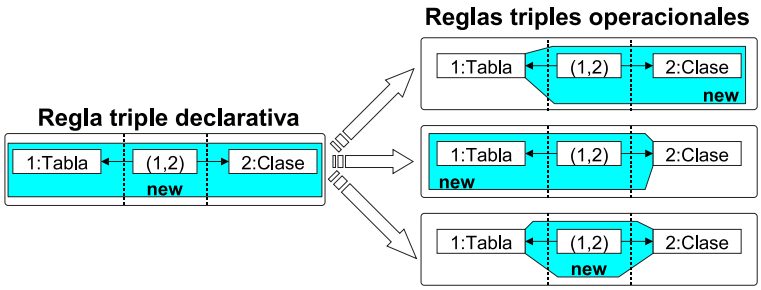


Figura 2.10: *Ejemplo de regla de gramática de grafos triples, y reglas operacionales*

correspondencia y los grafos origen y destino permitiendo morfismos a enlaces o dejándolos sin definir. Además, la monotonicidad puede resultar un grave inconveniente en cierto tipo de transformaciones, así como el hecho de no poder especificar condiciones de aplicación en las reglas.

2.3. Lenguajes visuales de dominio específico

Los Lenguajes de Dominio Específico (LDEs) son lenguajes que ofrecen abstracciones de alto nivel para la especificación de la estructura y el comportamiento en un dominio concreto. Suelen ser lenguajes pequeños que definen un conjunto de primitivas para representar conceptos del dominio. Esto permite a los expertos en el dominio un aprendizaje más rápido del lenguaje, así como de la especificación, creación y mantenimiento de sistemas definidos con ellos. Al trabajar con conceptos de alto nivel, quienes los usan pueden concentrarse en la esencia del problema y dejar de lado otros detalles accidentales, aumentando así su productividad, y mejorando la calidad, fiabilidad y facilidad de mantenimiento de los sistemas generados. En ocasiones, su uso es inevitable en aquellos dominios donde los lenguajes de propósito general no permiten representar adecuadamente los conceptos del dominio. Algunos ejemplos de LDEs son Mathematica en el dominio de la computación simbólica, o HTML para la especificación de hipertexto en el dominio web.

Cuando las primitivas que define el LDE son gráficas hablamos de Lenguajes Visuales de Dominio Específico (LVDEs). Los LVDEs aúnan las ventajas de cualquier LDE y las del uso de una notación visual. Al ser visuales resultan intuitivos y permiten que personas con conocimientos del dominio puedan usar el lenguaje para desarrollar nuevas aplicaciones, incluso sin tener conocimientos de programación. Los LVDEs se utilizan para el análisis y diseño de sistemas en diversos ámbitos, como por ejemplo la ingeniería web [37, 53, 163], las bibliotecas digitales [126], el modelado de formalismos híbridos [111], la programación paralela [22] o las redes de tráfico [184]. También se utilizan a menudo en procesos de desarrollo de software dirigido por modelos y en ingeniería de familias de productos para personalizar la variabilidad de una familia de sistemas [152, 188].

A la hora de diseñar un LVDE normalmente se diferencia entre la sintaxis abstracta y concreta. La primera incluye los conceptos del lenguaje y sus relaciones, y la segunda define la apariencia gráfica de los elementos de la sintaxis abstracta. La figura 2.11 muestra un ejemplo de autómata finito representado usando la sintaxis abstracta a la izquierda, y la sintaxis concreta a la derecha. En la concreta, los estados se representan como círculos con el nombre del estado dentro. Si el estado es inicial se muestra la cabeza de una flecha pegada al círculo, mientras que si es final se muestra un doble círculo. Las transiciones se representan como flechas adornadas con el símbolo asociado. Finalmente, las entradas al autómata y el puntero al estado actual se representan con cajas.

Existen diversos enfoques para especificar la sintaxis abstracta de un LVDE que clasificamos en declarativos y operacionales. Entre los declarativos, el primero de ellos consiste en utilizar un grafo de tipos con la definición de los conceptos y relaciones del lenguaje. Un grafo de tipos es similar a un meta-modelo pero no contiene cardinalidades ni restricciones. La segunda opción es definirlo con un meta-modelo. Los meta-modelos suelen expresarse usando lenguajes de alto nivel (como diagramas de clases UML o entidad-relación), y per-

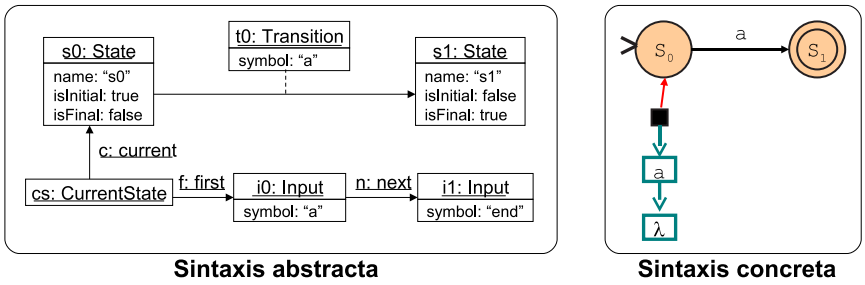


Figura 2.11: *Sintaxis abstracta y concreta del LVDE "Autómatas Finitos"*

miten definir cardinalidades y restricciones sobre los elementos del meta-modelo usando un lenguaje de restricciones (como OCL). La OMG propone otras dos alternativas declarativas para la definición de LVDEs, que son los perfiles UML y la extensión del meta-modelo UML [182]. Los perfiles permiten adaptar (pero no modificar) UML a las necesidades de un dominio de aplicación mediante la definición de estereotipos, valores etiquetados y restricciones. Los perfiles no extienden el meta-modelo de UML: aunque semánticamente la extensión se realiza en el nivel $M2$, formalmente se implementa en $M1$. Esto quiere decir que se pueden definir en cualquier entorno de modelado UML. Si el LVDE a definir es muy distinto de UML, otra opción es extender el meta-modelo de este último usando un entorno MOF. En ese caso la extensión se realiza en el nivel $M2$.

En cambio, los enfoques operacionales se basan en la definición de una gramática de grafos que, o bien verifica la corrección sintáctica de los modelos mediante su reducción a un símbolo inicial (lo que se conoce con el nombre de *parsing*), o bien modela las acciones que los usuarios pueden realizar para construir un modelo correcto (crear elementos, conectarlos, etc.). Ambos tipos de gramática son distintas implementaciones de un procedimiento para verificar la corrección de un modelo.

Cada uno de estos enfoques tiene sus ventajas e inconvenientes. Dentro de las propuestas declarativas, una de las ventajas de definir un LVDE usando un meta-modelo construido desde cero (esto es, que no sea un perfil ni una extensión de UML) es que proporciona una mayor flexibilidad a la hora de decidir la estructura, nombres, etc. de los elementos del meta-modelo. Además, permite adoptar cualquier sintaxis concreta para el lenguaje, mientras que usando perfiles la sintaxis concreta es la de UML (la cual puede resultar poco intuitiva en algunos dominios). Por otro lado, un perfil se puede construir con cualquiera de los entornos UML existentes, mientras que en el resto de enfoques se necesita construir un entorno nuevo para el lenguaje. Aún así, de todos los enfoques, los perfiles son los que tienen una menor riqueza expresiva. Por último, las extensiones de UML implican que el LVDE definido contendrá todo UML, lo cual puede ser una desventaja en algunas ocasiones. Respecto a los enfoques basados en transformación de grafos, a veces resultan más complicados de usar porque requieren definir una regla por cada situación posible.

Para definir la sintaxis concreta de un LVDE (esto es, cómo los conceptos y relaciones de la sintaxis abstracta se representarán gráficamente) existen varios enfoques. En el caso más sencillo, si la estructura de las dos sintaxis es similar, basta con asignar una representación gráfica a cada elemento de la sintaxis concreta. Sin embargo, en ocasiones es posible encontrar relaciones arbitrarias entre la sintaxis abstracta y la concreta. En general se pueden tener conceptos de la sintaxis abstracta que no tienen una representación gráfica, o que por el contrario pueden representarse de distinta forma dependiendo de ciertos factores (por ejemplo el tipo de diagrama donde aparece). También es posible encontrar casos donde ciertos elementos de la sintaxis concreta no están asociados a ningún elemento de la sintaxis abstracta. Para resolver estos problemas una posible solución es proporcionar definiciones separadas para la sintaxis abstracta y concreta, junto con las relaciones apropiadas entre sus elementos [86].

Los entornos visuales para la manipulación de LVDEs deben contemplar todos estos conceptos en su infraestructura. Las técnicas más habituales para su construcción son el desarrollo procedimental (codificar a mano la herramienta), el meta-modelado [51, 55, 63, 78, 118, 136, 195] y la transformación de grafos [18, 138]. El primero de ellos implica un alto coste en tiempo y recursos ya que requiere un gran conocimiento del API del lenguaje de implementación, así como del manejo de librerías gráficas. El meta-modelado [12] permite especificar la sintaxis del LVDE mediante un meta-modelo, y a partir del mismo se genera automáticamente un entorno para el lenguaje. Las herramientas de meta-modelado son fáciles de usar, y los entornos generados con ellas dan mucha flexibilidad a los usuarios a la hora de construir sus modelos. Finalmente, el uso de herramientas basadas en transformación es más complejo porque se necesita definir una regla por cada acción posible del usuario. Además, los entornos generados suelen ser más restrictivos ya que imponen un orden en la forma de construir los modelos. No obstante, tienen la ventaja de permitir modelar interacciones complejas. En muchas ocasiones lo que se hace es seguir un enfoque mixto con lo mejor de todos ellos.

2.4. Calidad del software

La calidad del software es "*el conjunto de características de una entidad que le confieren su aptitud para satisfacer las necesidades expresadas y las implícitas*" (ISO 9126) [98].

Por necesidades expresadas entendemos las que aparecen en los requisitos del sistema. La discordancia entre los requisitos y el producto final implican una falta de calidad. Para que un producto cumpla las necesidades expresadas debe ser correcto (debe satisfacer las especificaciones y hacer lo que éstas recogen), completo (debe satisfacer todos y cada uno de los requisitos), y fiable (las operaciones deben realizarse en todo momento de la forma esperada, sin fallos, y con la precisión requerida). Por necesidades implícitas entendemos aquellas que, aunque no se hayan especificado explícitamente o se hayan dado de manera incompleta, son deseables en el producto final y su falta puede acarrear una pérdida en la calidad del mismo. El rango de necesidades implícitas que pueden afectar a la calidad del producto final es bastante amplio. Como ejemplo podemos citar la eficiencia (las operaciones deberían tener un tiempo de respuesta adecuado), facilidad de uso, facilidad de mantenimiento (tanto para localizar y corregir posibles errores, como para modificar el comportamiento existente o añadir funcionalidades nuevas), alta cohesión (la estructura del sistema debería dividirse en módulos o componentes con una funcionalidad específica), reusabilidad (de los componentes desarrollados en posibles futuros proyectos), etc.

La ingeniería del software dispone de técnicas que permiten controlar y mejorar la calidad del software para satisfacer necesidades expresadas e implícitas en cada etapa del desarrollo. Sin embargo, es en las fases de análisis y especificación de requisitos donde mayor número de errores se cometen (del 65 % al 85 %) [23]. La codificación sólo introduce un 25 % de los errores y su verificación es la más costosa. Por tanto la calidad debe controlarse desde las etapas iniciales del desarrollo. Otro factor que refuerza esta observación es que el coste de reparar un error crece exponencialmente según el proyecto avanza hacia su conclusión [154]. En conclusión, encontrar y reparar los errores, e intentar obtener los mejores atributos de calidad desde las primeras fases de un proyecto software, puede ahorrar una gran cantidad de dinero y esfuerzo.

Para controlar la satisfacción de las necesidades expresadas, a veces se recurre al uso de métodos formales. Éstos pueden servir para definir de manera no ambigua lo que un sistema debe hacer y validar los requisitos (especificación formal), o para verificar que el producto final cumple las especificaciones originales y está ausente de errores (verificación formal). En ambos casos el objetivo es garantizar la calidad del producto final.

Para el control de las necesidades implícitas se necesitan criterios de valoración que establezcan cuándo algo tiene calidad y cuándo no la tiene. De hecho, la medición juega un papel central en la mayoría de las disciplinas de ingeniería. En ingeniería software, las métricas de producto [68] miden características de los sistemas software asociadas a necesidades implícitas (ej. facilidad de mantenimiento, cohesión, etc.) y establecen criterios

de calidad. Junto a mecanismos de medición, también se necesitan técnicas que guíen en la obtención de la calidad buscada. En ese sentido, los procedimientos más habituales son el uso de patrones de diseño [76] que establecen soluciones óptimas a problemas recurrentes en un dominio, o bien el rediseño [105] de la estructura interna de los modelos para mejorar sus propiedades. El *refactoring* de modelos [74, 133] es un ejemplo de rediseño que no cambia el comportamiento expresado en el modelo.

A continuación se da una introducción a las técnicas mencionadas (métodos formales, métricas, rediseños y patrones de diseño) para el control de la calidad del software.

2.4.1. Métodos formales para validación y verificación

Un lenguaje formal es un lenguaje con una sintaxis y semántica definidos de manera precisa mediante matemáticas o lógica. Ayuda a especificar de manera no ambigua sistemas software y hardware complejos. Un método formal está definido mediante un lenguaje formal más un sistema de razonamiento formal. Los métodos formales permiten razonar y derivar propiedades de los sistemas especificados con lenguajes formales, para de ese modo obtener sistemas de mayor calidad que cumplan los requisitos deseados[1].

La principal ventaja del uso de métodos formales es que su uso reduce el número de errores de un sistema, aumentando así su calidad. El problema es que esa mejora en la calidad se consigue a costa de un aumento en el coste de desarrollo. Para muchos sistemas el coste de aplicar métodos formales es demasiado alto, y se puede conseguir una calidad más que aceptable mediante inspección. Sin embargo, en sistemas de seguridad crítica donde el coste de fallo es muy alto, los métodos formales son indispensables para conseguir la corrección requerida y merece la pena el coste. De hecho, en muchos casos se puede demostrar que el incremento de los costes inducido por el uso de métodos formales es menor que la suma del coste del desarrollo original sin métodos formales más el coste de mantenimiento que es evitado mediante el uso de estos métodos. También hay que tener en cuenta que no todos los métodos formales incurren en el mismo coste de aplicación, siendo los de refutación los de menor coste, como ahora veremos. Sin embargo los métodos de refutación no realizan una verificación completa de la corrección del sistema: detectan la presencia de errores pero no su ausencia. En general, cada dominio de aplicación o sistema requerirá una solución específica.

Hace algunos años, el uso de métodos formales no estaba muy extendido debido a su dificultad de aprendizaje, baja escalabilidad y alto coste de aplicación. Sin embargo, hoy existen metodologías más eficientes y herramientas más potentes que han convertido los métodos formales en una técnica a tener en cuenta. En cuanto a su dificultad de aprendizaje,

[1]Los métodos formales no ayudan en el proceso de captura de requisitos, sino en la identificación de inconsistencias en su especificación. Si una especificación no recoge correctamente lo que se desea que el sistema haga, el código que se construya será consistente con la especificación pero no hará lo deseado.

la tendencia actual es ocultar su complejidad mediante la utilización de lenguajes de alto nivel, de tal modo que el usuario utiliza los lenguajes de alto nivel para especificar el sistema, internamente el sistema se verifica utilizando métodos formales de manera transparente al usuario, y los resultados se muestran nuevamente al usuario en la notación de alto nivel. De hecho, este será el enfoque utilizado en la presente tesis para la verificación de propiedades de sistemas especificados mediante LVDEs.

Hay tres grupos principales de métodos formales [23]: la verificación, el estudio matemático del factor clave de un problema, y la refutación.

Métodos formales de verificación. Proporcionan técnicas para demostrar (formalmente) si un sistema es consistente con una especificación dada. Existen dos aproximaciones distintas a la verificación formal de sistemas [42]: la verificación de modelos y la prueba de teoremas.

- La verificación de modelos es una técnica que consiste en construir un modelo de estados finito del sistema y comprobar si cumple la propiedad a verificar. Para ello realiza una búsqueda exhaustiva sobre el conjunto de estados que forman el modelo. Existen dos vertientes distintas. En la primera, denominada verificación de modelos temporal, el sistema se modela como un autómata finito de transición de estados y las especificaciones se expresan en lógica temporal. En la segunda vertiente, tanto el sistema como las especificaciones se modelan como autómatas, y lo que se hace es comprobar si el comportamiento de ambos es congruente. La verificación de modelos es un proceso completamente automático. Sin embargo tiene el inconveniente de la explosión de estados ya que, si el sistema es muy grande, el modelo de estados puede llegar a no poder almacenarse de manera explícita en un ordenador. En ese sentido existen diversas técnicas para minimizar el problema, como la representación simbólica de estados, la simetría, el uso de órdenes parciales, etc.
- En la prueba de teoremas, tanto el sistema como las propiedades a verificar se expresan en lógica matemática. Probar un teorema consiste en encontrar la prueba de una propiedad a partir del conjunto de axiomas y reglas de inferencia del sistema. Al contrario que los verificadores de modelos, es posible manejar un número de estados infinitos mediante técnicas inductivas.

En general, aunque los métodos formales de verificación son capaces de probar la corrección de todo un sistema en base a una especificación dada, pocas veces este tipo de pruebas se lleva a cabo completamente debido a su alto coste de aplicación.

Estudio matemático del factor clave. Surge con la intención de reducir el coste de aplicación de los métodos formales de verificación, y para ello se centra exclusivamente en el estudio matemático de los aspectos complejos de un sistema. Es decir,

en vez de especificar el sistema entero sólo se consideran las partes críticas. Como contrapartida a esta simplificación surge el riesgo de no considerar elementos que, sin embargo, sí influyen en los factores bajo estudio. Para comprobar qué partes del sistema no son relevantes para el estudio de un factor hay que recurrir de nuevo a los métodos de verificación formal. De todas formas, según se adquiere conocimiento en un determinado tipo de aplicación llega a ser fácil identificar cuáles son esas partes.

Métodos formales de refutación. En vez de demostrar que un sistema es correcto, intentan demostrar que no lo es. La ventaja de esto es que el coste de aplicación es mucho menor que el de los métodos de verificación formales, ya que para refutar la presunción de corrección basta con encontrar un contraejemplo, mientras que para demostrar su corrección es necesario considerar todos los casos posibles. Sin embargo, no debe olvidarse que este tipo de métodos pueden utilizarse para mostrar la presencia de errores, pero no su ausencia.

En el capítulo 5 se mostrarán algunos ejemplos de integración de métodos formales para la verificación de sistemas especificados en entornos de modelado. En particular, se estudiarán propiedades semánticas de diversos sistemas mediante su transformación al formalismo redes de Petri y posterior análisis con *model checking*, siendo éste uno de los métodos formales de refutación de uso más extendido. Ambos formalismos se presentan de manera intuitiva a continuación.

Redes de Petri

Las redes de Petri [145] son un formalismo matemático que Carl Adam Petri inventó en 1962 como generalización de los autómatas finitos deterministas. Facilitan expresar concurrencia, y permiten modelar de forma intuitiva y natural paralelismo, sincronización, no-determinismo, comunicación y recursos compartidos. En concreto, resultan especialmente útiles para modelar y analizar sistemas distribuidos y concurrentes.

Una red de Petri es un grafo dirigido bipartito formado por *lugares* (también llamados sitios, estados, o *places* en inglés) y *transiciones*. Las aristas del grafo se denominan *arcos* y conectan lugares a transiciones, o transiciones a lugares. Los arcos tienen un peso asociado, que es 1 por defecto. Los lugares contienen un número arbitrario de *marcas* (también llamadas puntos o *tokens* en inglés), cuya distribución en los lugares de la red da el *marcado* de la red. Típicamente, los lugares representan condiciones en el sistema y las transiciones representan eventos. Un lugar de entrada a una transición representa una pre-condición para que la transición tenga lugar, mientras que un lugar de salida representa una post-condición.

Gráficamente, los lugares de una red se representan como círculos, las transiciones como barras o rectángulos, y las marcas mediante puntos negros dentro de los lugares. Si

el número de marcas es grande, a veces se representan con un número dentro del lugar correspondiente. Por simplicidad, si el peso de los arcos es 1 no se muestra. Por ejemplo, la figura 2.13 muestra una red de Petri que modela una cola de procesos representados como marcas. Los procesos pueden estar esperando a ser despachados (lugar `Queue`), estar en proceso (lugar `Busy`) o haber finalizado (lugar `Exit`). El peso de los arcos no se muestra porque es 1. El marcado de la red está formado por dos marcas en `Queue` y una en `Busy`.

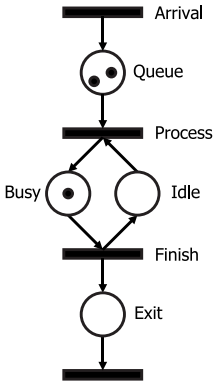


Figura 2.12: *Ejemplo de red de Petri*

La ejecución de una red de Petri consiste en una secuencia de disparos de sus transiciones. Una transición puede dispararse sólo si está activada, esto es, si todos los lugares conectado a ella mediante arcos de entrada tienen como mínimo el número de marcas especificado en el peso del arco (1 si no se indica otra cosa). Como resultado del disparo, a los lugares conectados a la transición mediante arcos de entrada se les resta tantas marcas como peso tenga el arco, mientras que a los lugares conectados mediante arcos de salida se les suma tantas marcas como indique el peso del arco. El disparo es una operación atómica sin pasos intermedios. La ejecución de una red de Petri es no-determinista ya que puede haber varias transiciones activadas simultáneamente, pudiendo disparar cualquiera de ellas.

La figura 2.13 muestra un ejemplo de disparo en la anterior red de Petri. La red de la izquierda corresponde al estado inicial del sistema, y tiene dos marcas en el lugar `Queue` y una en `Busy`. A su derecha se muestran coloreadas las dos transiciones que están activas en la red para ese marcado: `Arrival`, ya que no tiene ningún lugar de entrada (y por tanto cumple la condición sobre el número de marcas en los lugares de entrada) y `Finish`, ya que su único lugar de entrada `Busy` tiene una marca. Hay no-determinismo porque ambas transiciones pueden disparar. Suponiendo que disparase `Finish`, se restaría una marca de los lugares de entrada (`Busy`) y se sumaría una a los lugares de salida (`Idle` y `Exit`), obteniendo como resultado la red que se muestra más a la derecha.

Las redes de Petri tienen propiedades que proporcionan información sobre el sistema que están modelando. Algunas de estas propiedades son:

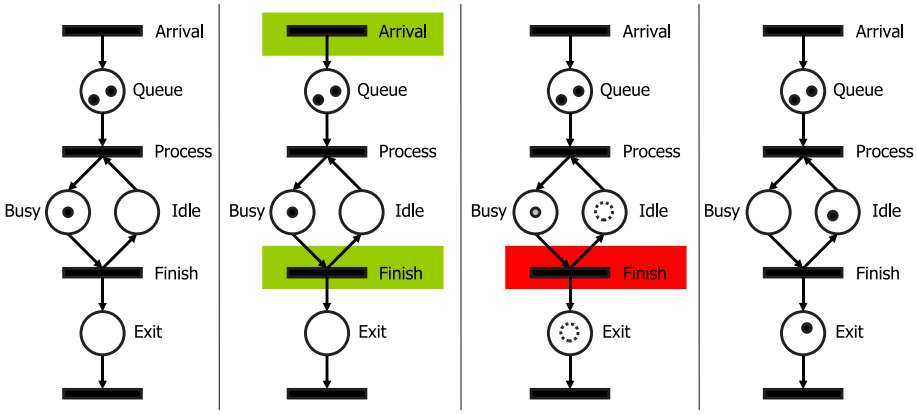


Figura 2.13: *Ejemplo de disparo de una red de Petri*

- *Alcanzabilidad.* Dada una red de Petri con un marcado inicial, decimos que cierto marcado m es alcanzable si existe una secuencia de transiciones que llevan desde el marcado inicial hasta m.

- *Vivacidad.* Una transición t es viva para un marcado inicial si existe una secuencia de disparos que contiene a t a partir de cualquier marcado sucesor del inicial. Una red de Petri es viva para un marcado inicial si todas sus transiciones son vivas, en cuyo caso la red no se bloquea nunca (no hay *deadlocks*). Existen distintos niveles de vivacidad para una transición según el número de veces que pueda dispararse (nunca, quizás, al menos k veces, siempre en cierta secuencia de disparos, o siempre para cualquier secuencia de disparos).

- *Exclusión mutua.* Dos lugares están en exclusión mutua para un marcado inicial, si no pueden estar marcados simultáneamente en ningún marcado sucesor del inicial. Esta propiedad puede utilizarse para detectar si existen tareas excluyentes.

- *Persistencia.* Una red de Petri es persistente si, para cualquier conjunto de transiciones habilitadas, el disparo de una no deshabilita el resto. El estudio de esta propiedad permite comprobar si distintos procesos paralelos pueden llegar a interrumpirse.

- *Acotación.* Una red de Petri está $k - acotada$ para un marcado inicial, si el número de marcas en cada lugar es menor o igual a k para todos los marcados sucesores del inicial.

- *Conservación.* Una red de Petri es conservativa para un marcado inicial, si la suma de marcas para cualquier marcado sucesor del inicial es un número entero constante.

- *Reversibilidad.* Una red es reversible si existe una secuencia de disparos que lleva al marcado inicial.

- *Invariantes.* Una ecuación sobre el número de marcas es invariante si se cumple durante todas las posibles ejecuciones.

Las redes de Petri definen diferentes métodos para estudiar estas propiedades, como son las técnicas algebraicas, estructurales, de reducción y el grafo de alcanzabilidad. Las técnicas algebraicas representan una red de Petri mediante un sistema de ecuaciones y utilizan técnicas algebraicas para su análisis. Permiten estudiar las condiciones necesarias de alcanzabilidad, invariantes, vivacidad y acotación de una red. Las técnicas estructurales analizan propiedades que dependen de la estructura de la red y no del marcado. Las técnicas de reducción buscan reducir la complejidad de una red preservando ciertas propiedades, esto es, a partir de una red construyen otra más pequeña con las mismas propiedades. Por último, el grafo de alcanzabilidad de una red de Petri es un grafo en el que los nodos representan el vector de marcado, y los arcos son las transiciones que, al dispararse, cambian el estado de la red. En aquellos casos en que el número de la red no es finito (debido a que la red no está acotada) se construye una aproximación a este grafo, denominado grafo de cubrimiento. La figura 2.14 muestra un ejemplo de red de Petri, y a la derecha su grafo de alcanzabilidad. El nodo inicial del grafo (señalado con una punta de flecha) contiene el marcado inicial de la red. El marcado se representa con un vector con tantas posiciones como lugares tiene la red, y donde cada posición indica el número de marcas que tiene el lugar asociado. En el ejemplo, dado el marcado inicial pueden disparar las transiciones `T1` y `T2`, lo que en cada caso nos lleva a un marcado distinto de la red ([0 1 0 0 1] o [1 0 0 1 0]). Para esos marcados se evalúa de nuevo qué transiciones podrían disparar (`T2` y `T4` respectivamente), lo que en ambos casos nos lleva de nuevo al estado inicial.

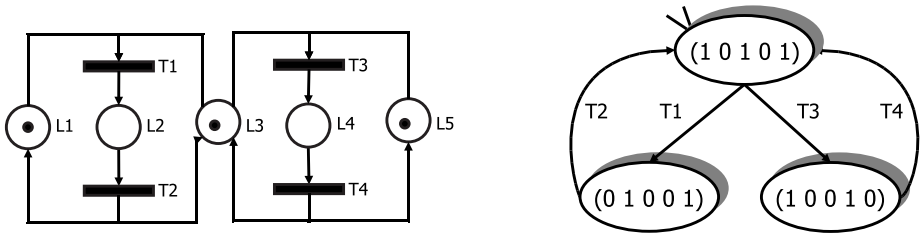


Figura 2.14: *Ejemplo de red de Petri y grafo de alcanzabilidad*

Una vez construido el grafo de alcanzabilidad o cubrimiento de la red es posible analizar distintas propiedades, como por ejemplo si un estado es alcanzable, la secuencia de transiciones que lleva a él, o comprobar si el sistema llega a un *deadlock* (situación en la que no hay transiciones activadas). Para expresar propiedades más complejas se puede utilizar lógica temporal, y su satisfacción se puede comprobar mediante la aplicación del algoritmo *model checking*. Dicho algoritmo se explica a continuación.

Model Checking

El algoritmo *model checking* [41] es una técnica automática para la verificación de sistemas reactivos finitos de estados. Las propiedades a verificar se expresan mediante lógica temporal, y el sistema se modela como un grafo de transición de estados. El mecanismo para comprobar si el grafo (o sistema) satisface las propiedades es, básicamente, una búsqueda eficiente sobre el mismo.

Existen distintas lógicas temporales que pueden utilizarse para expresar las propiedades de un sistema, como lógica de árboles de computación (*Computational Tree Logic*, CTL) o lineal (*Linear Temporal Logic*, LTL). Este apartado se centrará en CTL ya que el capítulo 5 presentará algunos ejemplos basados en su uso. Una fórmula CTL está formada por proposiciones atómicas que corresponden a propiedades que pueden o no cumplirse en cada estado del sistema. Las proposiciones se pueden combinar mediante conectores booleanos ($\wedge$, $\vee$, $\neg$), cuantificadores de caminos que expresan si los predicados se cumplen a partir de cierto estado, y operadores temporales que describen las propiedades de una rama del árbol de computación. Los cuantificadores de caminos son E (existe un camino) y A (para todos los caminos). Los operadores temporales básicos son X (en el siguiente estado) y U (hasta). Otros operadores temporales como F (en algún estado del futuro) y G (siempre) se pueden expresar en términos de X y U.

Los tipos de propiedades que se pueden verificar con los tipos de lógicas temporales más habituales son [127]:

- propiedades de seguridad del tipo "algo malo nunca pasa". Este tipo de propiedades suele expresar requisitos que el sistema debe cumplir en todo momento (invariantes). Por ejemplo, la expresión CTL $\neg E(TrueU(s_1 \wedge s_2))$ indica que los estados del sistema s_1 y s_2 no deben ocurrir simultáneamente en ningún camino de ejecución.

- propiedades de vivacidad del tipo "algo bueno pasa eventualmente". Estas propiedades representan requisitos que no tienen por qué mantenerse continuamente, pero cuya realización eventual (o continuada) se debe garantizar. Por ejemplo, la expresión CTL $A(TrueU(s_1 X s_2))$ indica que s_1 es inevitable, y después de s_1 siempre se produce s_2.

Model checking presenta dos ventajas frente a otros verificadores y demostradores de teoremas. La primera es que su coste de aplicación es bastante menor ya que no realiza una búsqueda exhaustiva de la corrección del sistema, sino que intenta encontrar una prueba de que ocurre lo contrario. La segunda ventaja es la alta automatización del proceso. Normalmente, el usuario proporciona una representación de alto nivel del modelo, así como la propiedad que desea verificar. El algoritmo devuelve si el modelo cumple la propiedad

o, en caso contrario, un contraejemplo que muestre por qué no se satisface. Estos contraejemplos son muy útiles para depurar sistemas complejos donde los errores sutiles podrían pasar inadvertidos.

Los primeros *model checkers* representaban los estados del sistema explícitamente, y por tanto eran incapaces de manejar sistemas grandes debido al problema de la explosión de estados. A finales de la década de los 80 aparece la variante conocida como *model checking simbólico*, que ya permite manejar sistemas reactivos grandes de hasta 10^{120} estados. La clave es representar el comportamiento del sistema mediante n variables booleanas, y cada transición en el sistema mediante una función booleana que relaciona dos estados del sistema consecutivos. Estas fórmulas se almacenan en árboles de decisión binarios [33] obteniendo así una representación concisa que no depende ni del número de estados del sistema ni del tamaño de la relación de transición. En la actualidad han surgido nuevas técnicas que, combinadas con métodos simbólicos, permiten el análisis de sistemas aún mayores. Dos de estas técnicas son el razonamiento composicional y la abstracción.

El razonamiento composicional [41] se usa en aquellos casos en que las especificaciones se pueden descomponer en propiedades locales de pequeñas partes del sistema. La estrategia de verificación consiste en comprobar cada una de esas propiedades locales usando sólo la parte del sistema que las define. Si podemos demostrar que el sistema satisface cada propiedad local y, además, la conjunción de propiedades locales nos da la especificación completa, entonces el sistema también satisface la especificación.

La segunda técnica, denominada abstracción, se basa en reducir la complejidad del sistema antes de aplicar *model checking*. Es aplicable a aquellos sistemas donde existe una relación sencilla entre el valor de los datos que se manejan (por ejemplo, una operación de adición en un microprocesador requerirá que el valor del registro de salida sea igual a la suma de los registros de entrada). El mecanismo se basa en establecer una relación entre los valores de los datos del sistema en el momento actual, y un pequeño conjunto de valores de datos abstractos. Si ese mecanismo se extiende al conjunto de estados y transiciones, se obtiene una versión abstracta del sistema bajo consideración. La versión abstracta suele ser mucho más pequeña que el sistema original, y por tanto es más sencillo verificar sus propiedades.

2.4.2. Métricas

Según [20, 191], las disciplinas de la ingeniería requieren mecanismos de medición que proporcionen información y ayuden en la evaluación, para de ese modo crear una memoria corporativa y ser capaces de responder a preguntas sobre lo medido. En ingeniería del software, los objetos que pueden ser medidos son los procesos, los recursos, los productos [68] y los proyectos [191]. Este apartado se centra en la medición de productos, y en particular en la medición de los modelos usados para la especificación de sistemas, ya que uno de los

objetivos de esta tesis es definir un mecanismo para incorporar herramientas de medición en entornos visuales para LVDEs.

Las métricas o medidas software son catalizadores para la obtención de productos de calidad. Se utilizan, por ejemplo:

- En ingeniería directa, para detectar anomalías y estimar el coste y esfuerzo requeridos para obtener un producto software. En particular, en el DSDM los modelos son entidades activas a partir de los cuales se genera código; por tanto, en este caso la calidad debería asegurarse a nivel de modelo. Es aquí donde se centran algunos objetivos de esta tesis.

- En ingeniería inversa, para adquirir un conocimiento básico del software con el objetivo de proporcionar vistas de alto nivel y encontrar violaciones de un buen diseño software.

- En evolución de software, para identificar qué partes del software son estables y cuáles inestables, localizar qué partes necesitan rediseñarse, e identificar variaciones en la calidad del software.

Existen distintas clasificaciones para métricas según el aspecto bajo estudio. Por ejemplo, dependiendo del tipo de atributo que estemos midiendo, podemos diferencias entre métricas *internas* y *externas*. Las métricas internas miden atributos internos del producto, es decir, características intrínsecas de éste. Por ejemplo, el número de líneas de código de un programa es un atributo interno del programa que nos da una medida de su tamaño. En cambio, las métricas externas miden atributos externos del producto, que son aquellos que sólo pueden medirse con respecto a cómo el producto se relaciona con el entorno [191]. Por ejemplo, la complejidad cognitiva, la usabilidad o la facilidad de mantenimiento son atributos externos. Este tipo de métricas se obtienen mediante pruebas y observando el software en ejecución.

Las métricas también se pueden clasificar en *directas* e *indirectas* en función de posibles dependencias para su cálculo. En las directas, el valor de las métricas se deriva del valor de algún atributo que no depende del resultado de otra medición. Por ello a veces se las agrupa bajo el nombre de medidas base [77]. Las métricas indirectas (o derivadas) se obtienen de la combinación de varias métricas directas o indirectas. El término *indicador* se utiliza a veces para referirse a métricas indirectas que tienen asociado un modelo de análisis. Éste es un procedimiento de cálculo con unos criterios de decisión, que pueden ser un conjunto de valores umbral, objetivos o patrones usados para determinar cuándo se necesita realizar una acción (por ejemplo, un rediseño) o realizar una investigación más profunda [77].

Desde el punto de vista de su objetividad, podemos diferencias entre aquellas en cuyo cálculo interviene el juicio humano (subjetivo), y las que son cuantificaciones basadas en re-

glas numéricas (métodos objetivos). Finalmente, dependiendo del grafo de automatización, los métodos de medida pueden ser automáticos, semi-automáticos o manuales.

2.4.3. Rediseños y patrones de diseño

Los rediseños son cambios en la estructura de un modelo de diseño con objeto de mejorar algún aspecto de su calidad, como por ejemplo la comprensibilidad, rendimiento, cohesión o acoplamiento. Cuando el rediseño preserva la semántica se denomina *refactoring* de modelos [133]. El término *refactoring* [74] se acuñó para definir las modificaciones realizadas en código fuente y hacerlo más legible y/o fácil de mantener sin cambiar el comportamiento observable. El *refactoring* de modelos intenta elevar el nivel de abstracción de esta técnica y aplicarla sobre los modelos que describen los sistemas.

Para detectar si un modelo debe someterse a un rediseño se suele recurrir a los llamados "malos olores" (del inglés *bad smells* [74]). Éstos describen de manera informal problemas habituales en código o diseño, y proponen una serie de acciones que pueden ayudar a su corrección. Existen trabajos recientes para la definición formal de tales "olores" a través de métricas [144].

Por otro lado, un patrón de diseño [76] recoge el conocimiento y la experiencia de múltiples expertos en un determinado área para aportar soluciones que han demostrado ser óptimas en la resolución de problemas recurrentes. De este modo, los diseñadores pueden aliviar la toma de decisiones recurriendo a soluciones contrastadas y validadas empíricamente. Las ventajas derivadas del uso de patrones de diseño son que pueden mejorar la productividad y calidad de las soluciones software gracias a que promueven la reutilización del diseño y del conocimiento, forman un vocabulario común de diseño entre los participantes en el proceso de desarrollo, y mejoran la documentación [38].

Algunos de estos patrones de diseño son específicos de dominio, y por tanto están descritos en términos cercanos al dominio [107, 114, 190]. En esos casos, aunque su campo de aplicación es específico, abordan problemas de un mayor nivel de abstracción ya que sus soluciones reflejan estructuras conceptuales de un dominio de conocimiento.

Debido a su éxito el uso de patrones se ha extendido a otros campos, y de ese modo encontramos patrones de análisis para el modelado de sistemas [73], patrones de procesos [8] o patrones de arquitecturas [34].

2.5. Conclusiones

El capítulo ha presentado algunos conceptos teóricos básicos para la comprensión de este trabajo: LVDEs, meta-modelado, transformación de modelos y calidad del software. Los LVDEs son notaciones gráficas restringidas a un dominio concreto y aportan beneficios en términos de calidad y productividad. Constan de una sintaxis abstracta con los conceptos del dominio y de una sintaxis concreta que asigna una apariencia gráfica a cada concepto. La sintaxis abstracta se puede especificar mediante un meta-modelo, a partir del cual se genera un entorno de modelado para el lenguaje. Si el lenguaje está formado por una familia de notaciones o tipos de diagrama (como ocurre en UML) se le denomina LVDE multi-vista.

Para manipular modelos (que pueden estar especificados mediante un LVDE) se usan los llamados lenguajes de transformación de modelos. La transformación de grafos es un ejemplo de lenguaje de transformación de modelos de carácter declarativo, formal y visual. Al ser formal proporciona técnicas para analizar propiedades de las transformaciones como su terminación o confluencia. Al ser declarativo y visual permite expresar manipulaciones de modelos de manera intuitiva.

Por último, el capítulo ha mostrado algunas técnicas útiles para controlar la calidad de un modelo, o más correctamente, del sistema que representa un modelo de diseño. Para verificar la corrección, completitud o fiabilidad se pueden utilizar métodos formales. Como ejemplo se estudiaron el formalismo redes de Petri y el algoritmo *model checking*. Otras propiedades de calidad como la complejidad o cohesión de un sistema pueden controlarse mediante el uso de métricas que las cuantifiquen, o de rediseños que modifiquen el modelo para mejorar alguna característica de calidad.

Los objetivos de esta tesis buscan dotar a los entornos de modelado para LVDEs de herramientas y mecanismos integrados para el control de la calidad desde los siguientes puntos de vista: verificación de sistemas (en particular para la gestión de la consistencia entre diagramas y el análisis de propiedades), medición (automática y objetiva de los sistemas expresados mediante el uso de LVDEs) y rediseños. Estas herramientas deben generarse preferiblemente a partir de descripciones gráficas de alto nivel para facilitar la labor del desarrollador del entorno visual. En particular se usarán técnicas de meta-modelado, transformación de grafos, LVDEs y patrones para conseguir este propósito.

Capítulo 3

Estado del arte

Este capítulo muestra una panorámica de los entornos visuales existentes desde dos perspectivas: cómo se generan, y cómo integrar herramientas para controlar la calidad de los modelos especificados mediante su uso. El capítulo se divide en seis secciones. La primera presenta los principales enfoques para la definición de LVs multi-vista, lo que incluye mecanismos de consistencia entre vistas, análisis de propiedades semánticas, y obtención de vistas derivadas en respuesta a una consulta.

La segunda sección recoge los enfoques existentes para la generación de entornos para LVDEs, entre los que destacan el meta-modelado y la transformación de grafos. También se analizan las herramientas más representativas de cada enfoque centrándose en qué dispositivos proporcionan para definir mecanismos de control de calidad que se integren en los entornos generados, ya sea para la definición de lenguajes multi-vista, mecanismos de consistencia de la semántica estática y dinámica entre diagramas, técnicas de análisis y verificación, simuladores, o medidas y rediseños específicos de dominio.

Ya que la propuesta para la especificación de LVDEs realizada en esta tesis hará uso de transformación de grafos, la sección 3.3 presenta algunos de los lenguajes de transformación más utilizados. Para cada lenguaje se presentan sus características, para qué tipo de transformación son adecuados, y qué herramientas hay disponibles para su uso.

La sección 3.4 estudia algunas herramientas CASE que integran mecanismos de validación y verificación de los modelos de un sistema software, o bien que usan herramientas externas para este propósito. Con ello se pretende deducir el tipo de análisis que habitualmente se requiere en una herramienta de modelado. Por otro lado, ya que el capítulo 5 presentará algunos ejemplos donde se usa *model checking* como mecanismo interno de análisis, la sección también presenta diversas herramientas de verificación basadas en este algoritmo y el tipo de propiedades que permiten verificar.

A continuación, la sección 3.5 presenta propuestas de modelos genéricos para la especificación de métricas y rediseños independientes del lenguaje y/o dominio, así como enfoques formales para la especificación de medidas y rediseños. La sección incluye un apartado sobre herramientas CASE que permiten la medición y reestructuración de modelos, así como otras herramientas de medición y rediseño genéricas. Al igual que antes, el propósito es deducir qué mecanismos para la medición y manipulación de modelos específicos de dominio requiere un entorno de modelado.

Para finalizar, la última sección da una breve introducción al desarrollo dirigido por modelos y analiza algunas herramientas utilizadas en este paradigma de desarrollo. Este estudio se centra en las características (de aseguramiento de la calidad) relevantes para esta tesis. La razón de incluir esta sección es doble: por un lado, la generación de entornos visuales basada en meta-modelado se puede considerar un caso particular de desarrollo dirigido por modelos, y por tanto resulta interesante conocer los conceptos que este paradigma define; por otro lado, los objetivos que esta tesis pretende cubrir pueden servir como base al desarrollo dirigido por modelos, mediante la generación de entornos visuales avanzados para LVDEs con el soporte de calidad adecuado que este paradigma necesita.

3.1. Enfoques para la especificación de lenguajes multi-vista

Como introdujo la sección 2.3, existen dos enfoques principales para definir un LV-DE: el meta-modelado y la transformación de grafos. En ambos casos lo que se hace es proporcionar una definición de la sintaxis del lenguaje (un meta-modelo o una gramática, respectivamente), la cual se usa para validar si un modelo dado es válido en el lenguaje. Como puede observarse, ambas técnicas permiten verificar la sintaxis de un modelo pero no analizar su relación con otros modelos del sistema. Por el contrario, especificar un LV-DE multi-vista requiere definir no sólo la sintaxis del tipo de diagramas que el lenguaje comprende (quizás usando uno de estos enfoques básicos), sino también especificar mecanismos adicionales que garanticen o verifiquen su consistencia sintáctica y semántica. A continuación se muestra el estado del arte de los principales enfoques para la definición de lenguajes multi-vista, incluyendo la obtención de vistas derivadas. El estudio se realiza desde un punto de vista teórico. Posteriormente, la sección 3.2 realizará un estudio práctico evaluando diversas herramientas para generar entornos visuales, y estableciendo cuáles proporcionan soporte para la definición de este tipo de lenguajes.

3.1.1. Vistas de un sistema. Definición y consistencia

El concepto Punto de Vista como mecanismo para integrar múltiples perspectivas en el desarrollo de un sistema apareció originalmente en [69]. En este trabajo, un punto de vista es un agente con conocimiento parcial sobre el sistema. Tiene un estilo (la notación usada), un dominio (el área de conocimiento), una especificación, y un plan de trabajo. El plan de trabajo contiene las acciones disponibles para construir la especificación, y también acciones de control para verificar la consistencia sintáctica y de la semántica estática de uno o varios puntos de vista. En concreto, la consistencia se gestiona mediante la definición de relaciones que deben existir entre los puntos de vista, y evaluando posteriormente la existencia de dichas relaciones mediante acciones de control. Estas reglas de consistencia se definen textualmente dando los puntos de vista a verificar y la relación que debe existir para que la regla se evalúe positivamente [57].

Posteriormente han surgido muchas y muy diversas propuestas para la especificación y consistencia de puntos de vista, algunas de las cuales se analizan a continuación. Esto ha llevado a su definición en el estándar 1471 del IEEE [95]. En este estándar, una vista (del sistema) se define como la representación de un sistema desde la perspectiva de un conjunto de intereses. Un punto de vista es una descripción o plantilla desde la cual desarrollar vistas, y establece su propósito, a quién va dirigida, técnicas de creación y mecanismos de análisis. En enfoques basados en meta-modelado el mecanismo de creación de un punto de vista sería el meta-modelo, pero aún se necesitaría especificar otra información adicional para

definir completamente el punto de vista.

Otro estándar reciente que también recoge la noción de vista es el lenguaje QVT (*Queries/Views/Transformations*) [155] definido por la OMG. En QVT una vista es un modelo derivado de un modelo base del cual o bien es un subconjunto, o bien contiene la misma información pero reorganizada para una tarea o usuario concreto. Una vista no se puede modificar separadamente del modelo desde el que se deriva, sino que cualquier cambio realizado en el modelo base debe propagarse a la vista (y viceversa si la vista es editable). QVT contempla la definición de vista como una parte de un meta-modelo, lo que corresponde al concepto de punto de vista en el estándar 1471 del IEEE. Aunque la especificación actual de QVT aún no implementa un mecanismo para la definición de vistas, está previsto que se realice con transformación de modelos para permitir la propagación de cambios requerida.

Mecanismos de consistencia estática

La consistencia de la semántica estática (o sintáctica) entre vistas de un sistema incluye la satisfacción de restricciones sintácticas inter-diagrama y la propagación de cambios (esto es, coherencia de la información que aparece replicada en distintas vistas). La mayoría de enfoques existentes en la literatura dan soluciones para notaciones específicas (*Open Distributed Processing* (ODP) [30, 56], UML, notaciones web, XML [183], etc.) bien describiéndolos con un formalismo, bien especificando un conjunto de reglas de consistencia específicas, o realizando extensiones del lenguaje para permitir la especificación de tales reglas. También se encuentran muchas herramientas que implementan mecanismos de consistencia para lenguajes específicos. Sin embargo, los enfoques generales para la consistencia inter-modelo son más escasos. Existen diversas propuestas basadas en diversos formalismos, como lógica de predicados de primer orden [175] o gramáticas de grafos [79, 91].

Por ejemplo, el enfoque propuesto en [175] se basa en proporcionar un modo común de describir los distintos puntos de vista. Para ello definen un *grafo conceptual* con todos los elementos que pueden aparecer en los puntos de vista, y especifican después con qué concepto del grafo conceptual se identifican los elementos de cada punto de vista. Al crear una vista se obtiene su grafo conceptual, el cual se transforma a lógica de predicados de primer orden para verificar la consistencia.

En [91] los autores definen un mecanismo para construir transformaciones que respeten o establezcan la consistencia entre modelos. Para ello primero se deben identificar las situaciones de inconsistencia que no deben ocurrir en los modelos. Esto se hace mediante condiciones de consistencia gráficas (GCCs), que son grafos con tipo y atributos que pueden contener condiciones de aplicación negativas. A partir de estas GCCs crean reglas de transformación que contienen como parte izquierda una GCC (esto es, una inconsistencia) y como parte derecha un grafo que soluciona la inconsistencia.

En [79] utilizan transformación de grafos distribuida para formalizar la integración de las

diversas vistas de un sistema. Un grafo distribuido es un grafo en el que los nodos contienen grafos (denominados sistemas locales) y las aristas entre nodos contienen morfismos que establecen relaciones entre los elementos de los sistemas locales. Los objetos de un sistema local pueden hacerse accesibles a otros sistemas mediante interfaces. Para modificar un grafo distribuido se utilizan producciones sincronizadas. De este modo, la especificación de un sistema software se puede representar como un grafo distribuido donde los nodos del grafo son las diferentes vistas, las aristas establecen dependencias entre ellas, y las reglas distribuidas implementan la propagación de cambios.

Mecanismos de consistencia dinámica

Existen tres grandes enfoques para verificar la semántica dinámica (y en general para el análisis) de las vistas de un sistema:

1. El primer enfoque consiste en especificar las vistas utilizando un lenguaje formal (Z [170], Maude [43], etc.) con mecanismos de análisis que permitan verificar su consistencia. Sin embargo, el proceso de especificar las vistas es costoso y requiere conocimientos especializados que no son frecuentes entre la media de ingenieros.

2. El segundo enfoque, que constituye la tendencia actual, es el uso de métodos formales ocultos. En este enfoque las vistas del sistema se especifican utilizando una notación intuitiva (como UML, LVDEs, etc.) y después se traducen automáticamente a un formalismo donde realizar los análisis de interés. En este caso, o bien las vistas se integran en un modelo único que es el que se transforma al formalismo, o bien cada vista se transforma por separado y son los modelos resultantes los que se deben integrar. Ejemplos prácticos de este enfoque para un lenguaje origen fijo se pueden encontrar en [92, 121, 175, 184, 186, 192], muchos de ellos orientados a la validación de modelos UML. El rango de formalismos elegidos como destino de la transformación es muy amplio, y abarca por ejemplo redes de Petri [121, 184] o CSP [92]. Algunas propiedades típicas que suelen verificarse con este tipo de enfoque son la ausencia de bloqueos de los recursos de un sistema, la imposibilidad de que ocurran situaciones no deseadas, o comprobar si el sistema permite la realización de una determinada tarea.

   Como puede suponerse, para que los resultados resulten útiles deben mostrarse de nuevo en la notación original. Sin embargo, aunque este enfoque se utiliza con mucha frecuencia, existen pocas técnicas generales para la anotación de resultados desde el dominio semántico a la notación original. En [186] se propone usar grafos de referencia donde la vista del sistema y su equivalente en el formalismo se componen en un solo grafo que contiene relaciones auxiliares 1-a-1 entre los elementos de ambos modelos. Esto es distinto de un grafo triple, donde hay tres grafos independientes y ningún

elemento auxiliar. Dado un conjunto de elementos en el formalismo, la anotación se obtiene siguiendo las relaciones auxiliares en el grafo de referencia. El mismo mecanismo podría utilizarse realizando la transformación con QVT [155], ya que ésta crea trazas entre los modelos fuente y destino. En ambos casos el mecanismo de anotación es implícito y consiste en seguir un conjunto de enlaces o trazas. Esto tiene dos limitaciones importantes: cada elemento en el formalismo sólo puede anotarse a un elemento en el modelo origen, y no es posible anotar resultados que no sean representables en el modelo original.

3. Un tercer y último enfoque para la consistencia dinámica es la simulación, la cual permite estudiar la evolución del valor de las variables de un sistema. En ocasiones esto incluye la animación de los modelos para de ese modo facilitar la verificación del comportamiento de manera gráfica, viendo cómo los elementos se "mueven". Para sincronizar la simulación de las vistas de un sistema, algunos autores construyen con ellas un modelo único que es el que simulan. Otro enfoque es la co-simulación [71], donde cada tipo de vista se anima usando un simulador específico, y la interacción entre vistas se resuelve en tiempo de simulación. Finalmente algunos autores optan por una simulación multi-formalismo, donde todas las vistas se transforman a un formalismo único para el que se realiza la simulación [122, 185].

   En ocasiones los simuladores se construyen expresando la semántica operacional del lenguaje en un dominio semántico (esto es, se usan métodos formales ocultos para simulación en vez de para análisis). Al igual que antes, el resultado de la simulación debe mostrarse en términos del lenguaje original y no del dominio semántico. Para ello, en [66] extienden las reglas operacionales del dominio semántico con los elementos del lenguaje que se quiere animar. Este enfoque permite simular pero no verificar modelos. En [19] emparejan las reglas de creación del lenguaje y las del dominio semántico. Posteriormente, cada paso de simulación en el dominio semántico se traslada al lenguaje original usando un lenguaje textual de programación, dando como resultado la animación de los modelos expresados en el LV.

3.1.2. Vistas derivadas

Una vista derivada es el modelo resultante de realizar una consulta sobre un modelo base. Por ejemplo, las vistas de una base de datos pueden considerarse un caso particular de vista derivada obtenida como resultado de una consulta sobre una base de datos. Sin embargo una base de datos no es un LVDE, y por tanto los lenguajes de consulta para este tipo de entidades quedan fuera del alcance de este estudio. Hecha esta aclaración, debe decirse que existen muchas propuestas de lenguajes de consulta para grafos y modelos que se clasifican según:

1. la sintaxis del lenguaje de consulta, que puede ser textual [176] o visual [17, 100, 173].

2. el modo de construir la vista, distinguiendo entre *operacionales* si la vista se construye a partir de un conjunto de reglas, o *declarativos* [17, 173] si se construye a partir de una descripción de las características que los elementos de la vista derivada deben cumplir. Los lenguajes declarativos son de más alto nivel que los operacionales ya que constituyen una descripción compacta de un conjunto de reglas (esto es, las reglas operacionales se derivan a partir de la descripción declarativa). Si además la consulta se expresa mediante patrones resultan intuitivos, ya que los patrones usan la misma notación que usa el modelo consultado.

   Dentro de los enfoques operacionales se pueden incluir cualquiera de los lenguajes habituales para la transformación de modelos (véase sección 3.3), entre los que hay textuales y visuales. Aunque el objetivo de estos lenguajes no es la obtención de vistas derivadas, se podrían usar con este propósito.

   Entre los enfoques declarativos hay diversas propuestas para expresar consultas de manera visual mediante patrones. Por ejemplo, VIATRA2 [17] permite expresar consultas en grafos mediante patrones de grafos recursivos que pueden contener patrones positivos y negativos anidados de profundidad arbitraria. Este tipo de patrones tiene la potencia expresiva de una lógica de primer orden [156]. Otra propuesta basada en patrones, pero para consultas sobre modelos basados en MOF, es JPDD [173]. Este lenguaje permite expresar consultas utilizando una notación similar a UML que incluye símbolos especiales para operaciones de selección. La semántica de esas expresiones está definida mediante meta-operaciones OCL que al ejecutarse sobre un modelo UML obtienen todos los elementos que cumplen tales expresiones.

   No obstante, no hay muchos enfoques declarativos visuales que ofrezcan mecanismos para sincronizar el modelo base y la correspondiente vista derivada respecto a futuros cambios en los mismos.

3. la estructura del modelo obtenido, que puede ser *con materialización* si los elementos de la vista derivada son duplicados del modelo base, o *sin materialización* [100] si el resultado es una vista lógica cuyos elementos son los del modelo base. El primer tipo de lenguajes es más lento ya que implica crear los elementos del resultado. Es más, como el resultado es un modelo nuevo, si se necesita mantenerlo consistente con el modelo base hay que proporcionar mecanismos de propagación de cambios. Esos mecanismos no se necesitan en el segundo tipo de lenguajes ya que, en ese caso, los elementos de la vista derivada son los del modelo base (aunque en cualquier caso habría que actualizar qué elementos pertenecen a la vista). En cambio, en el segundo tipo de lenguajes se requieren mecanismos de bloqueo adicionales (similares a los usados en bases de datos) para evitar la modificación simultánea del mismo elemento

desde varias vistas. Además, a veces se necesita trabajar con el resultado de una consulta sin que los cambios se vean reflejados inmediatamente en el sistema (por ejemplo si la vista se va a usar para hacer pruebas o simulación). En conclusión, el primer tipo de lenguajes es apropiado si no se quiere sincronización o ésta sólo debe realizarse desde el modelo base a la vista, mientras que el segundo tipo es más eficiente si se necesita que el modelo base y las vistas estén sincronizadas en todo momento.

En [100] muestran cómo obtener vistas sin materialización utilizando una versión extendida de gramáticas de grafos triples denominada VTGG. Su objetivo es la integración de herramientas sin que haya duplicación de datos. Para ello realizan una implementación del patrón adaptador de clases (*class adapter*) para obtener un solo grafo, el adaptador de la vista, cuyos objetos implementan las interfaces de la vista y las interfaces del modelo base. Así se evita duplicar objetos del grafo y no hay que propagar modificaciones.

Recientemente, la importancia de disponer de lenguajes para realizar consultas sobre modelos ha quedado patente con la OMG y su lenguaje QVT [155]. En QVT, la parte *query* tiene como objetivo definir consultas sobre modelos. Con este propósito algunos autores han propuesto utilizar OCL [176] ya que resulta apropiado para entornos de meta-modelado basados en MOF (no así para entornos de meta-modelado no basados en MOF, o sistemas basados en transformación de grafos, véase sección 3.2). OCL es textual y tiene una gran potencia expresiva. Sin embargo, el resultado no es gráfico (obtiene el conjunto de clases y relaciones que cumplen la consulta, y para visualizarlo en forma de modelo habría que proporcionar un mecanismo adicional). Para facilitar la construcción de consultas complejas, QVT incluye una operación denominada *helper* que encapsula secuencias de expresiones OCL (extendido). Si la operación no tiene efectos laterales se dice que es una consulta. Por otro lado, QVT también permite seguir un enfoque operacional basado en transformación de modelos para obtener las vistas derivadas, para lo que habría que definir el conjunto de reglas de transformación necesarias. En este caso la vista se materializaría como un modelo separado. Además, las trazas resultantes de la transformación permitirían sincronizar el modelo base y la vista derivada mediante propagación de cambios bidireccional.

3.2. Generación de entornos para lenguajes visuales de dominio específico

Los LVDEs son lenguajes gráficos diseñados para su uso en un área de aplicación concreto, y proporcionan estructuras gráficas de alto nivel que representan conceptos del dominio. Desde un punto de vista práctico, resulta útil disponer de entornos donde construir modelos escritos en un LVDE (que puede ser multi-vista) dado. Esos entornos pueden ayudar a reducir el tiempo de modelar un sistema y el número de errores cometidos en su especificación. La funcionalidad básica de este tipo de entornos debe garantizar la corrección sintáctica de cada modelo construido. Adicionalmente, algunos entornos avanzados integran herramientas de control de la calidad de los modelos que cubren uno o varios de los siguientes aspectos: consistencia entre diagramas; análisis, verificación o simulación de los modelos; medición de características relevantes de los mismos; y reestructuración automática de los modelos para mejorar su calidad.

Dada la complejidad asociada al desarrollo de tales entornos, se suele recurrir a técnicas automáticas para generarlos a partir de una descripción de alto nivel del LVDE. En la actualidad los dos enfoques más usados son el meta-modelado y la transformación de grafos. Aunque existen otras soluciones aparte de éstas, su uso no está tan generalizado. Esta sección presenta las ideas básicas de los distintos enfoques existentes y algunas de las herramientas que los implementan. Se pondrá especial interés en los mecanismos que cada herramienta proporciona para definir no sólo la sintaxis del LVDE, sino también cualquier mecanismo orientado a mejorar la calidad de los modelos construidos en los entornos generados. Debe aclararse que sólo se considerarán los mecanismos que constituyan una ayuda real para el desarrollador del entorno visual (esto es, no se tendrá en cuenta la posibilidad de codificarlos a mano desde cero).

3.2.1. Entornos visuales basados en meta-modelado

En los entornos basados en meta-modelado, la sintaxis abstracta del LVDE se define mediante un meta-modelo, y (en el caso más sencillo) a cada elemento del meta-modelo se le asigna una apariencia visual o sintaxis concreta. El meta-modelo puede incluir restricciones para reducir el conjunto de modelos válidos del lenguaje. Los meta-modelos suelen definirse utilizando una notación de alto nivel, como por ejemplo diagramas de clases UML. El entorno generado a partir de esa definición comprueba que los modelos construidos en el entorno son conformes al meta-modelo, para lo que debe existir un morfismo desde el modelo al meta-modelo, y además se deben cumplir las restricciones sobre el modelo.

A continuación se realiza una revisión de algunas de las principales herramientas de meta-modelado existentes. La primera de ellas, AToM^3, se utilizará en los capítulos 4 y 5 para implementar y evaluar el marco de especificación de LVDEs propuesto en esta tesis.

AToM^3

AToM^3 [51] (2002) es una herramienta de meta-modelado que se desarrolló originalmente para describir y generar entornos para lenguajes de simulación de sistemas dinámicos. La herramienta se basa en la idea de que "todo es un modelo", en el sentido de que incluso la interfaz de usuario de la herramienta es un modelo que se interpreta al cargarse y que, por tanto, se puede modificar para eliminar o añadir botones con nuevas funcionalidades.

En AToM^3, la sintaxis abstracta del LVDE se especifica mediante un meta-modelo que puede ser un diagrama de clases o un diagrama entidad-relación. El meta-modelo puede incluir restricciones escritas en Python (lenguaje en el que está implementado la herramienta). Para cada restricción se debe especificar qué evento de usuario causa su evaluación (crear un elemento, borrar un elemento, salvar el modelo, etc.) así como si la evaluación se realiza antes o después de que el evento tenga lugar. Por último, cada elemento del meta-modelo hereda de una clase padre que define un atributo para especificar su sintaxis concreta (un icono en el caso de clases y entidades, y una flecha en el caso de asociaciones y relaciones). La figura 3.1 muestra el uso de AToM^3 para definir un LVDE. La ventana del fondo contiene el meta-modelo del lenguaje, y la ventana superior corresponde a la edición de la sintaxis concreta de una de sus clases.

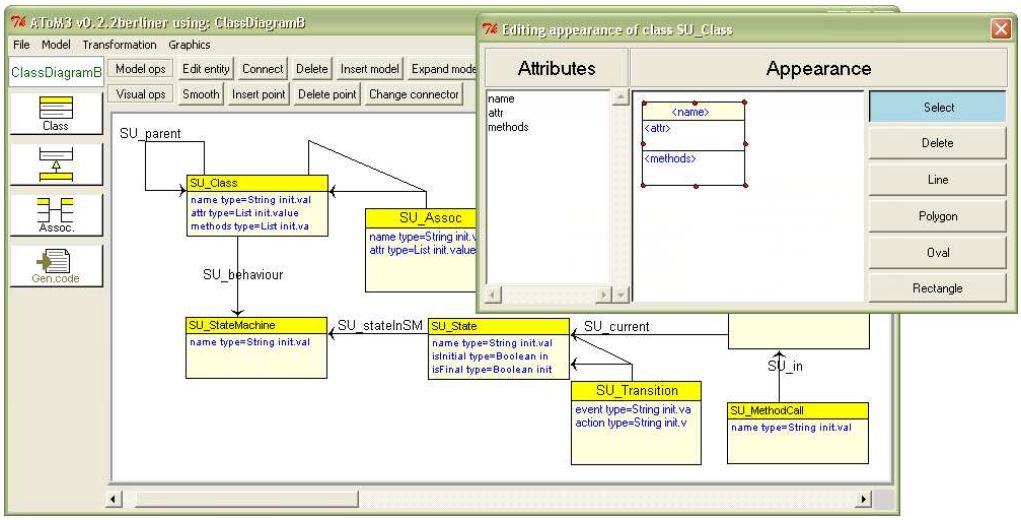


Figura 3.1: *Definición de un lenguaje visual de dominio específico en AToM^3*

Una vez definido, el meta-modelo se carga de nuevo en AToM^3 de tal forma que el comportamiento de la herramienta cambia y permite dibujar modelos que se ajustan a la sintaxis especificada. Por ejemplo, la figura 3.2 muestra el entorno generado para el lenguaje de la figura 3.1, donde los botones en la parte izquierda de la interfaz corresponden a elementos de su meta-modelo.

AToM^3 no permite definir LVDEs multi-vista, sino que habría que generar un entorno

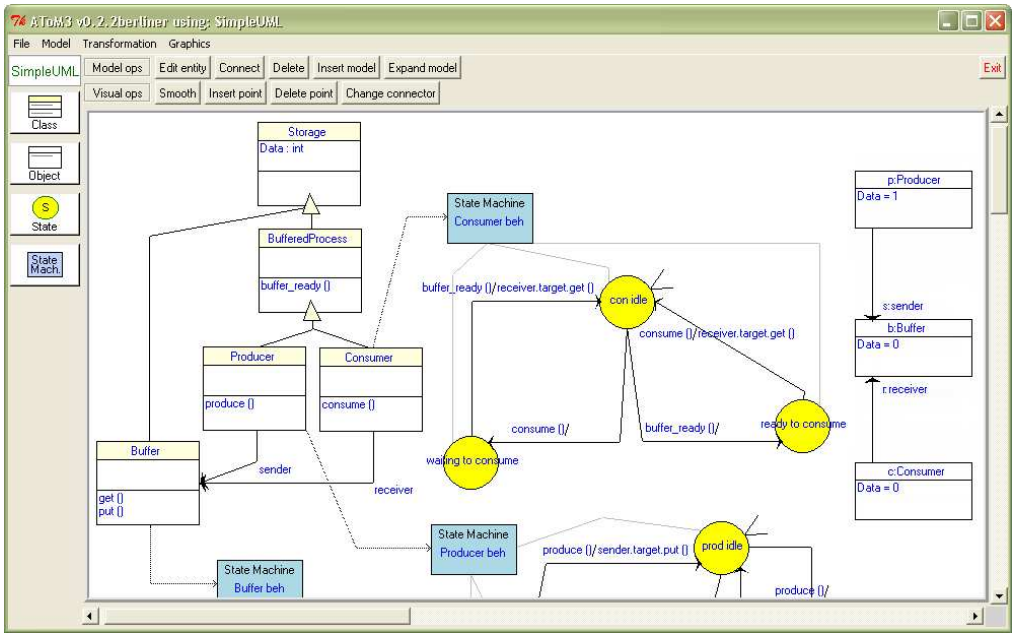


Figura 3.2: *Entorno generado para un lenguaje visual de dominio específico en AToM*^{3}

visual distinto e independiente para cada lenguaje de la familia. Tampoco incluye ni genera mecanismos de consistencia ni análisis entre diagramas. Los modelos se pueden manipular mediante gramáticas de grafos, lo que permite construir simuladores y especificar rediseños para un lenguaje dado. Sin embargo no incluye soporte para identificar en qué partes del modelo aplicar los rediseños, ni para especificar mecanismos de medición de los modelos del lenguaje. Tampoco permite obtener vistas derivadas.

El capítulo 4 mostrará algunas extensiones realizadas a AToM^3 (dentro de este trabajo de tesis) para eliminar las limitaciones enumeradas. La nueva versión de la herramienta sí que permite definir mecanismos de consistencia, análisis, medición y rediseño que se integran en el entorno de modelado generado, así como obtener vistas derivadas de un modelo. Para ello implementa las ideas propuestas en el marco para la especificación de LVDEs que presenta el mismo capítulo.

DSL Tools

DSL Tools [55] (2005) es la propuesta de Microsoft para crear entornos para LVDEs (y definir generadores de código para los mismos). La sintaxis abstracta del lenguaje se describe mediante un meta-modelo expresado con una variante de MOF, denominado modelo de dominio en la herramienta. Se pueden expresar restricciones sobre el meta-modelo utilizando código C#, las cuales se evalúan en respuesta a eventos de usuario. La sintaxis concreta consiste en la definición de iconos y conectores gráficos asociados a cada elemento

del meta-modelo, para lo que se utiliza un fichero XML. A partir de esta herramienta, genera un editor para el lenguaje visual.

Esta herramienta no ofrece soporte para ninguna de las características bajo estudio.

GME

GME (*Generic Modeling Environment*) [118] (1999) es una herramienta de meta-modelado para la creación de entornos de modelado de dominio específico y la síntesis de programas. La sintaxis abstracta de los LVDEs se especifica con diagramas UML, y la semántica estática se define con el lenguaje de restricciones OCL. Para especificar la sintaxis concreta, los elementos de los diagramas UML incluyen atributos para definir el nombre de un fichero de iconos o los colores y tipo de línea a usar para su representación.

A partir de esta descripción se genera un entorno que puede usarse para construir modelos del lenguaje, los cuales se almacenan en una base de datos o en formato XML. A partir de esos modelos se pueden generar automáticamente las aplicaciones, y sintetizar entradas a diferentes herramientas COTS y simuladores (desarrollados separadamente).

Respecto a las características avanzadas bajo estudio, GME permite definir modelos jerárquicos (modelos dentro de modelos), múltiples aspectos que definen qué partes de un modelo están visibles en cada momento, y composición de meta-modelos a través de las clases con igual nombre. La consistencia de la semántica estática entre modelos en aquellos casos en que un mismo elemento puede aparecer en distintos modelos se gestiona mediante la definición de referencias desde el elemento referenciado a los elementos que lo referencian. Un elemento referenciado no puede eliminarse hasta que todas sus referencias se han eliminado, y no hay posibilidad de cambiar tal patrón de comportamiento (por ejemplo, para implementar un borrado en cascada al eliminar el objeto referenciado). GME permite manipular los modelos con el lenguaje de transformación de modelos GreAT [6], lo cual permitiría construir simuladores y rediseños. Sin embargo no incluye soporte para identificar en qué partes del modelo aplicar los rediseños, verificar la consistencia de la semántica dinámica entre diagramas, ni realizar mediciones de los modelos.

GMF

GMF (*Graphical Modeling Framework*) [78] (2006) es una herramienta de meta-modelado construida sobre Eclipse que permite generar editores gráficos para lenguajes visuales. Los editores construidos se basan en EMF (marco de modelado y generación de código para especificar meta-modelos y gestionar sus instancias) y GEF (marco de edición gráfica).

Para especificar un LVDE en GMF se debe dar un meta-modelo con su sintaxis abstracta, otro con su sintaxis concreta, y un modelo que asigna objetos de la sintaxis concreta a la sintaxis abstracta (es decir, asigna visualizaciones a los conceptos del dominio). Para editar la sintaxis concreta no existe un editor gráfico, sino que los iconos se definen como

nodos de un árbol que contienen (recursivamente) el nombre de los objetos gráficos de los que está compuesto. Es posible especificar restricciones en OCL. A partir de esta definición se genera un entorno de edición de modelos conforme a la sintaxis del lenguaje, los cuales se almacenan en formato XMI o en cualquier otro esquema XML definido por el usuario.

La consistencia de la semántica estática entre modelos se soporta mediante sincronización de modelos. Aunque GMF no provee soporte para ninguna otra de las características bajo estudio, algunas se podrían implementar con otras herramientas Eclipse. Por ejemplo se pueden usar ATL [14], MTF [143] o VIATRA [17] para definir simuladores y rediseños mediante transformación de grafos, o EMF Query [65] para la obtención de vistas derivadas. Esta última herramienta proporciona un API para realizar consultas sobre modelos, cuyo resultado es el conjunto de objetos lógicos que cumplen la consulta (y no un modelo). Sin embargo habría que encargarse de la integración de las herramientas, lo que incluye mecanismos para reflejar el resultado de las operaciones realizadas sobre los modelos definidos en el entorno de modelado. En cualquier caso no hay soporte para el análisis de la semántica dinámica, la especificación de mediciones, ni la identificación de las partes del modelo donde aplicar los rediseños.

MetaEdit+

MetaEdit+ [136] (1991, primera versión comercial en 1995) es una herramienta que permite desarrollar entornos para LVDEs. El enfoque que sigue es distinto de las demás herramientas de esta sección ya que para definir el lenguaje no hay que dar un meta-modelo con su sintaxis abstracta. En cambio, lo que se tiene que hacer es modelar por separado los cinco conceptos que especifica el lenguaje de ingeniería de notaciones GOPRR, que son: los símbolos que usará la sintaxis concreta de los elementos del lenguaje; los atributos de los elementos; las relaciones entre elementos; los roles o tipo de elementos permitidos a ambos extremos de cada relación; y las reglas sintácticas que establecen qué elementos, relaciones y roles contiene el lenguaje y cómo pueden conectarse.

A partir de esta definición crea un entorno para la edición de modelos en la notación dada. Los modelos se pueden editar mediante diagramas gráficos, matrices o tablas.

MetaEdit+ es una herramienta multi-proyecto que permite intercambiar datos entre diversos proyectos, siendo el concepto de lenguaje multi-vista un caso particular de multi-proyecto. La consistencia de la semántica estática entre diagramas se logra mediante el uso de un repositorio de objetos común, de tal modo que al confirmar los cambios en un diagrama éstos se reflejan automáticamente en el resto de diagramas. Respecto al resto de características bajo estudio, el único soporte que MetaEdit+ proporciona es un lenguaje de script para exportar los modelos a distintos formatos (código, XML, etc.) que pueden ser entrada de herramientas de análisis, medición, transformación o simulación que deben desarrollarse por separado, y desde las cuales no hay mecanismos para anotar los resultados

sobre los modelos originales. Por tanto, desde la perspectiva adoptada en este estado del arte no proporciona un soporte real para ninguna de las características estudiadas.

Pounamu

En Pounamu [195] (2004), los entornos para LVDEs se definen mediante el uso de cuatro componentes: un editor de iconos, conectores y atributos; un editor de manejadores de eventos para especificar comportamiento en respuesta a eventos (restricciones) en Java; un editor de meta-modelos usando diagramas entidad-relación; y un editor para asignar iconos y conectores (sintaxis concreta) a los elementos del meta-modelo. A partir de esta definición genera un editor de modelos que se ajustan a la sintaxis especificada.

Pounamu soporta la definición de lenguajes multi-vista. Para cada vista se deben especificar los iconos, conectores y atributos que pueden aparecer en la vista, las entidades y relaciones del meta-modelo asociadas a la vista, y las relaciones entre los elementos del meta-modelo y los componentes de la vista. La consistencia de la semántica estática entre diagramas se genera automáticamente, y no es modificable. Otras restricciones más complejas se tienen que programar en Java. La herramienta no da soporte para el resto de características bajo estudio.

Tiger

Tiger [63] (2004) es un generador de editores de lenguajes visuales que se implementan como *plugins* para Eclipse. Al igual que GMF (explicado anteriormente) se basa en EMF y GEF para la definición de los lenguajes, pero facilita las tareas de definición de modelos de dominio, símbolos gráficos y relaciones entre ellos. Internamente Tiger utiliza el motor de transformación de grafos AGG [5] que opera sobre la sintaxis abstracta del modelo para generar el editor. Actualmente sólo permite definir lenguajes visuales sencillos basados en grafos, como por ejemplo diagramas de actividad o redes de Petri.

Respecto a las características bajo estudio, las consideraciones son las mismas que para GMF pero sin proporcionar sincronización de modelos. Como ventaja añadida, cuenta con un editor de gramáticas de grafos integrado en la herramienta para la manipulación de modelos.

3.2.2. Entornos visuales basados en transformación de grafos

Los entornos basados en transformación de grafos se generan a partir de una gramática que puede ser de creación o de análisis sintáctico (*parsing*). La primera clase de gramáticas da lugar a entornos dirigidos por sintaxis, donde cada regla de la gramática representa una posible acción del usuario en el entorno, y el usuario debe seleccionar en cada caso la regla que quiere aplicar. La segunda clase de gramáticas (de análisis sintáctico) intenta

reducir el modelo a un símbolo inicial para verificar la corrección del mismo. Ambas clases de gramáticas son implementaciones de un procedimiento para verificar la validez de un modelo, y por tanto, pese a estar basadas en reglas declarativas, son más procedimentales que los enfoques basados en meta-modelado.

GenGEd

GenGEd [18] (1999) es un entorno de desarrollo para lenguajes visuales especificados mediante la definición de un alfabeto y de una gramática de grafos sobre el mismo. Alfabeto y gramática son la entrada de un editor de diagramas basado en reglas y restricciones, que permite la manipulación dirigida por sintaxis de diagramas visuales mediante la aplicación de las reglas de la gramática.

En la actualidad el desarrollo de GenGEd se ha abandonado y se ha sustituido por Tiger, lo que hace pensar que un enfoque basado en meta-modelado es más adecuado para definir entornos visuales. Sin embargo, también es cierto que definir la sintaxis del lenguaje mediante una gramática puede ser más adecuado para acciones de edición complejas. Otra ventaja es que permite definir simuladores, rediseños y otras manipulaciones de modelos utilizando el mismo lenguaje de transformación usado para definir el lenguaje. Por último, también es posible definir gramáticas que guíen al usuario hacia la construcción de soluciones inspiradas en patrones de diseño.

Aparte de para actividades que implican la manipulación de modelos, GenGEd no da soporte para ninguna de las otras características bajo estudio.

DiaGen

DiaGen [138] (1995) permite desarrollar editores de diagramas especificando su sintaxis mediante gramáticas de hipergrafos. Al igual que ocurre con GenGEd, DiaGen ha evolucionado a una herramienta de meta-modelado llamada DiaMeta [139] (2006) que permite especificar lenguajes usando la implementación de MOF que proporciona MOFLON [141].

En general, las aportaciones y limitaciones de DiaGen sobre el conjunto de propiedades que se están estudiando son las mismas que para GenGEd.

3.2.3. Otros enfoques

CIDER

CIDER [101] (2004) es una herramienta basada en componentes para construir editores de diagramas "inteligentes" (esto es, editores que se comportan como si entendieran el significado de los diagramas construidos). Los editores se construyen a partir de un conjunto de componentes Java predefinidos. La sintaxis del lenguaje (no se diferencia entre sintaxis

concreta y abstracta) se especifica mediante un conjunto de símbolos gráficos más una gramática multiconjunto atribuida que impone restricciones gráficas entre los símbolos. Las reglas de estas gramáticas son textuales.

Es posible especificar manipulaciones de modelos para controlar la disposición de sus elementos, implementar simuladores y realizar razonamientos sobre los diagramas. También podrían usarse para especificar rediseños o guiar la construcción según patrones de diseño. Las transformaciones se especifican mediante reglas de producción textuales que pueden contener cuantificadores existenciales y contexto negativo.

No hay soporte para la definición de lenguajes multi-vista, mecanismos de análisis, medidas, ni identificación de submodelos candidatos a rediseño.

IPSEN

IPSEN [106] (1997) persigue la generación de entornos para lenguajes textuales y visuales (gráficos). Para ello, la sintaxis concreta y abstracta de cada lenguaje se especifica mediante gramáticas EBNF, a partir de las cuales se genera un editor dirigido por sintaxis para la sintaxis concreta textual. Adicionalmente se pueden definir anotaciones sobre los términos de las gramáticas EBNF y asignarles una representación gráfica, a partir de la cual generar el correspondiente editor para la sintaxis concreta visual. Para especificar manipulación de modelos se pueden usar sistemas de reescritura programados utilizando el lenguaje PROGRES.

No hay soporte para la consistencia entre diagramas, la definición de medidas, ni la identificación de partes de los modelos candidatas a rediseño.

3.3. Lenguajes de transformación de modelos

Al trabajar con modelos es habitual tener que manipularlos, por ejemplo para realizar rediseños automáticos, transformarlos a un formalismo distinto para su análisis, refinarlos, animarlos, etc. Existen muchos lenguajes para la transformación de modelos, y se pueden clasificar atendiendo a tres características ortogonales: si son visuales o textuales; si son imperativos, declarativos o híbridos (esto es, utilizan construcciones imperativas y declarativas); y si son formales o semi-formales. Esta sección recoge una breve descripción de los lenguajes de transformación de modelos más utilizados en la actualidad. A modo de resumen, la tabla 3.1 muestra los lenguajes estudiados, clasificados según sus propiedades.

	Textual	Visual	Imperativo	Declarativo	Híbrido	Formal	Semi-Formal
ATL	•				•		•
G.Grafos		•		•		•	
G.G.Triples		•		•		•	
GReAT		•			•		•
MOLA		•	•				•
QVT	•	•			•		•
RubyTL	•				•		•
Story d.	•	•			•	•	
VIATRA2	•	•			•	•	

Tabla 3.1: *Algunos lenguajes de transformación de modelos*

ATL

ATL (*ATLAS Transformation Language*) [14] es un lenguaje de transformación modelo-a-modelo textual e híbrido, basado en QVT (ver más abajo). Permite obtener un conjunto de modelos destino a partir de un conjunto de modelos origen, cada uno de ellos conforme a un meta-modelo dado. Cada transformación consta de reglas que al ejecutarse dan como resultado un modelo nuevo independiente del inicial. Admite dos tipos de transformación dependiendo de la similitud entre los modelos origen y destino: si van a ser muy distintos se debe especificar cómo obtener cada elemento del modelo destino a partir del modelo origen, pero si van a ser similares basta con especificar únicamente cómo obtener los elementos que difieren, y el resto se copia sin cambios desde el origen al destino. Nótese que el segundo tipo de transformación implica crear una copia del modelo origen, lo cual puede no ser óptimo en modelos grandes, y no es adecuado para algunas manipulaciones intra-formalismo como la animación de modelos.

Existe una implementación de ATL en la plataforma Eclipse, lo que permite su integración con otras herramientas Eclipse para MDA.

Gramáticas de grafos

Las gramáticas de grafos son un lenguaje visual, declarativo y formal para la transformación de modelos. Se basan en el uso de reglas, para cuya definición y aplicación existen diversas formalizaciones (véase apartado 2.2.1). Dada su naturaleza formal, muchos lenguajes de transformación de modelos como VIATRA2 [17] o los *story diagrams* [70] las usan como base teórica. Tales lenguajes suelen ser híbridos. Utilizan reglas de transformación de grafos para la manipulación de los modelos, y un lenguaje imperativo (textual o visual) para controlar la ejecución de las reglas o incrementar su expresividad.

Existen diversas herramientas que implementan gramáticas de grafos, tales como AGG [5] o AToM^3 [51]. Entre ellas, AGG es la única que proporciona mecanismos para verificar propiedades de las transformaciones. AGG es un entorno de programación visual basado en transformación de grafos con tipo y atributos. El comportamiento de la aplicación se describe mediante reglas de grafos que modifican la estructura de un grafo dado. Las reglas siguen el enfoque algebraico DPO de transformación de grafos, y pueden contener condiciones de aplicación positivas y negativas. Además se pueden agrupar en capas que se ejecutan secuencialmente, mientras que dentro de cada capa las reglas se aplican no determinísticamente. Como técnicas de análisis, AGG proporciona el análisis de pares críticos para estudiar la confluencia de una gramática, y la verificación de restricciones condicionales en un grafo dado. En concreto, el análisis de pares críticos sobre una gramática obtiene todas las posibles aplicaciones de dos reglas que pueden dar lugar a un conflicto (esto es, los grafos mínimos tales que al aplicar dos reglas en distinto orden se obtiene un resultado distinto, véanse técnicas de análisis de gramáticas de grafos en el apartado 2.2.1 para más información).

Gramáticas de grafos triples

Las gramáticas de grafos triples son un lenguaje visual, declarativo y formal que resulta adecuado para la transformación modelo-a-modelo. Permiten manipular grafos triples compuestos por un grafo origen (de la transformación), un grafo destino, y un grafo correspondencia que contiene nodos que relacionan los elementos de los grafos origen y destino (véase apartado 2.2.1).

Entre las herramientas que soportan gramáticas de grafos triples destaca la herramienta de meta-modelado MOFLON [141]. Esta herramienta permite definir relaciones de equivalencia entre los elementos de dos meta-modelos mediante reglas triples declarativas. A partir esas reglas se genera código Java para la integración de modelos conforme a dichos meta-modelos, en concreto para crear un modelo a partir de otro, crear enlaces de correspondencia entre dos modelos dados, o propagar valores de atributos entre dos modelos.

GReAT

GReAT (*Graph Rewriting and Transformation*) [6] es el lenguaje de transformación de modelos disponible en el entorno de meta-modelado GME. Es visual y consta de un lenguaje de especificación de patrones, un lenguaje declarativo de transformación de grafos con una sintaxis similar a UML, y un lenguaje imperativo de control de flujo de alto nivel para controlar el orden de aplicación de las reglas. El lenguaje de control permite especificar secuencias de reglas, ejecución no determinista (esto es, ejecución de reglas en paralelo), composición de reglas, recursión y condiciones del tipo "si-entonces".

Para especificar una transformación modelo-a-modelo con GReAT hay que proporcionar los meta-modelos de los modelos origen y destino, más un tercer meta-modelo adicional con las relaciones auxiliares o de equivalencia existentes entre los elementos de los meta-modelos. Esto es similar a las trazas de QVT o a los nodos de correspondencia en gramáticas de grafos triples (véanse apartados correspondientes), aunque más expresivo ya que permite definir no sólo relaciones entre los elementos de los modelos origen y destino, sino también nuevas clases y asociaciones auxiliares. Al terminar una transformación se eliminan los elementos auxiliares de tal modo que quedan dos modelos conformes a sus respectivos meta-modelos.

MOLA

MOLA (*MOdel transformation LAnguage*) [142] es un lenguaje de transformación de modelos visual e imperativo. Combina programación estructurada con reglas de transformación basadas en patrones. En MOLA, una transformación consiste en una secuencia de declaraciones (en el caso más básico reglas) que se representan gráficamente como cajas unidas por flechas. Una regla contiene un patrón y una acción, ambos expresados en un único diagrama de objetos UML. Estos patrones permiten definir estructuras de control avanzadas de manera visual, como bucles para ejecutar una regla varias veces, o condiciones "si-entonces". También es posible llamar a otras reglas pasando como parámetro referencias a las instancias identificadas por la regla de llamada. La transformación se ejecuta sobre el modelo original (transformación *in-place*), y por tanto es más adecuado para transformaciones endógenas.

La herramienta *MOLA Tool* soporta las principales estructuras de control del lenguaje. Actualmente se está desarrollando una segunda versión que soporte el lenguaje completo.

QVT

QVT (*Queries/Views/Transformations*) [155] es el estándar propuesto por la OMG para la transformación de modelos en MDA, y por tanto compatible con otros estándares OMG como OCL o MOF. Es un lenguaje híbrido semi-formal que tiene una sintaxis concreta

visual y otra textual. Define tres lenguajes de transformación diferentes llamados *Relations*, *Core* y *Operational* que se diferencian en su naturaleza declarativa o imperativa, y en que sólo el primero de ellos tiene una sintaxis visual. Una transformación QVT recibe un conjunto de modelos origen y un conjunto de modelos destino, todos conformes a metamodelos MOF, y establece el conjunto de relaciones de equivalencia que deben existir entre ellos. Las relaciones declarativas pueden incluir cláusulas con condiciones adicionales y llamadas a otras reglas, estableciendo un flujo de control en la ejecución. Por último, es posible definir transformaciones de verificación (*checkonly*) que simplemente verifican la consistencia entre modelos, así como de modificación (*enforce*) que cambian uno de ellos para hacerlos consistentes.

Los usos de QVT son variados. Se puede utilizar tanto para verificar la consistencia de dos modelos como para forzarla. Permite realizar transformaciones modelo-a-modelo uni- y bi-direccionales, en el último caso mediante la creación de trazas entre los elementos de los modelos origen y destino. También se puede usar para establecer relaciones entre modelos pre-existentes y realizar actualizaciones incrementales de un modelo.

Existen bastantes trabajos recientes que intentan dotar de una semántica formal a QVT. Por ejemplo, el marco MOMENT [25] proporciona un conjunto de operadores genéricos sobre modelos (ej. la unión) definidos con el formalismo de especificación algebraica Maude. Esos operadores se utilizan como motor de reescritura para la transformación de modelos, lo que permite probar características de la transformación como su confluencia o terminación. Otro ejemplo es [109], donde proponen utilizar gramáticas de grafos triples para formalizar QVT. La idea es beneficiarse del parecido existente entre la parte declarativa de QVT y las gramáticas triples para realizar su integración.

Recientemente han surgido numerosas herramientas que soportan QVT (o una parte del mismo). Sin embargo la mayoría sólo tiene en cuenta la sintaxis textual [25, 132, 168], y no hay ninguna que soporte completamente su sintaxis visual.

RubyTL

El lenguaje de transformación modelo-a-modelo RubyTL [44] está implementado como un LDE embebido dentro del lenguaje de programación dinámico Ruby. Es textual e híbrido, donde la parte declarativa está formada por reglas y relaciones entre los elementos origen y destino de la transformación, y la parte imperativa viene especificada en Ruby. Proporciona un mecanismo de extensión de las características básicas del lenguaje mediante la creación de *plugins*. De este modo se puede extender tanto la sintaxis básica del lenguaje, como el algoritmo de transformación de modelos utilizado, o el ciclo de ejecución de las reglas. Por ejemplo, algunas características del lenguaje implementadas como extensiones son la invocación explícita de reglas, la ejecución de una regla un número de veces, o la agrupación de reglas en fases que se ejecutan secuencialmente (similar al concepto de

capas en transformación de grafos).

Story diagrams

Los *story diagrams* [70] son un lenguaje visual basado en gramáticas de grafos para la transformación de modelos. Suele usarse para modelar la evolución dinámica de un sistema (transformación de un modelo) más que para transformaciones modelo-a-modelo. Utiliza diagramas de clases UML para especificar la sintaxis de los modelos a manipular, diagramas de actividad UML para representar gráficamente estructuras de control, y diagramas de colaboración UML para especificar las reglas de reescritura de grafos. Las actividades de un *story diagram* pueden contener código o reglas de reescritura de grafos basadas en Progres.

El lenguaje está disponible en la herramienta de desarrollo dirigido por modelos Fujaba, el cual genera código Java a partir de los diagramas de clases y *story diagramas*.

VIATRA2

VIATRA2 [17] es un lenguaje de transformación de modelos cuyo objeto es servir de soporte al desarrollo de sistemas basados en modelos con la ayuda de métodos formales invisibles (esto es, verificación de modelos mediante su proyección a un formalismo). Es híbrido y dispone de una sintaxis concreta textual (VTML) para transformaciones grandes, y de otra gráfica para desarrolladores familiarizados con MOF/UML. El lenguaje integra transformación de grafos (de naturaleza formal) con máquinas de estados abstractas que proporcionan estructuras de control imperativas sobre el orden de ejecución de las reglas de transformación. Las reglas son patrones definidos como grafos con restricciones adicionales. VIATRA2 tiene una gran riqueza expresiva, y contempla por ejemplo patrones negativos anidados, patrones recursivos, reglas genéricas (esto es, reglas que tienen tipos como argumentos) y meta-reglas que modifican reglas. Aunque los conceptos que utiliza VIATRA2 no son estándar, planean dar soporte a estándares como QVT mediante *plugins* Eclipse.

Para realizar transformaciones modelo-a-modelo utilizan el concepto de grafo de referencia [186], que consiste en componer los modelos origen y destino en un solo grafo que contiene relaciones 1-a-1 entre los elementos de ambos modelos. Este enfoque es similar al seguido por GReAT o QVT. Existe una herramienta de soporte para VIATRA2 integrada en Eclipse.

Como puede verse, en la actualidad hay disponible una gran variedad de lenguajes de transformación de modelos de los cuales este apartado ha recogido sólo los de uso más extendido. Cada lenguaje presenta características particulares que lo hacen apropiado para la resolución de cierto tipo de problemas, por lo que antes de decantarse por uno de ellos debe estudiarse qué se pretende conseguir con su uso.

De los lenguajes estudiados, sólo ATL y RubyTL son meramente textuales. Ambos crean un modelo independiente como resultado y son menos apropiados para manipular la estructura de un modelo (transformaciones *in-place*).

Entre los lenguajes visuales, los *story diagrams*, QVT, VIATRA2, MOLA o GReAT tienen mayor riqueza expresiva que la transformación de grafos y grafos triples, ya que los primeros complementan el uso de reglas con lenguajes imperativos que guían su ejecución y permiten el paso de parámetros. Sin embargo QVT, MOLA y GReAT son semi-formales. Entre todos los lenguajes visuales los más apropiados para transformaciones *in-place* son los *story diagrams*, las gramáticas de grafos y MOLA, aunque no lo son tanto para transformaciones modelo-a-modelo. Para ese tipo de transformaciones son más adecuados GReAT, VIATRA2 y gramáticas triples, los cuales se basan en la definición de un meta-modelo o modelo único con relaciones auxiliares entre los modelos origen y destino para realizar la transformación modelo-a-modelo. Sin embargo de los tres sólo VIATRA2 y gramáticas triples tienen una base formal. En QVT tales elementos auxiliares (trazas) forman parte del lenguaje y no hay que definirlos. En cuanto a herramientas, QVT es el único lenguaje del que no se dispone de una herramienta que soporte completamente la sintaxis visual.

Respecto a la menor riqueza expresiva de la transformación de grafos y grafos triples frente a otros lenguajes visuales, cabe señalar el esfuerzo que diversos autores están realizando para su mejora (reglas abstractas [60], reglas recursivas [85], etc.).

3.4. Análisis y verificación de software

Para detectar si existen errores en un diseño o verificar su ausencia, los entornos de modelado pueden usarse en combinación con herramientas de verificación externas o pueden integrar en su interfaz mecanismos de verificación específicos. En el primer caso, analizar el sistema requiere cambiar a una herramienta distinta, traducir los modelos del sistema a la notación usada por la herramienta de verificación (probablemente un lenguaje formal), conocer el tipo de análisis requerido para cada propiedad que se desee verificar, y proyectar los resultados del análisis sobre los modelos de diseño. En el segundo caso los mecanismos de análisis vienen integrados en la herramienta de modelado, y por tanto están adaptados al lenguaje utilizado y tipo de análisis requerido, son automáticos, y en el mejor de los casos no requieren conocer ningún lenguaje adicional al usado por la herramienta.

Esta sección recoge una muestra de las herramientas de modelado del segundo tipo, esto es, que integran mecanismos de verificación y análisis en su interfaz. Como se verá, en muchas ocasiones los enfoques seguidos se basan en el uso de herramientas de verificación externas, pero su uso es transparente al usuario.

La sección concluye con un análisis de las principales herramientas de verificación basadas en *model checking* [41], por ser éstas unas de las más utilizadas para la verificación de sistemas.

3.4.1. Entornos de modelado con capacidad de análisis y verificación

Los principales enfoques de análisis o verificación de sistemas integrados en entornos de modelado son dos: basados en simulación y basados en transformación. El primer enfoque consiste en proporcionar simuladores que permitan verificar empíricamente el diseño de un sistema mediante la animación de los modelos que lo forman. El segundo enfoque consiste en transformar internamente los modelos de diseño al lenguaje de entrada de alguna herramienta de verificación (interna o externa) donde se realiza el análisis. En este enfoque los resultados a veces se dan sobre el modelo original, pero en otros casos se devuelve lo obtenido por la herramienta de verificación y debe ser el usuario quien interprete el resultado.

Dentro del primer enfoque, un ejemplo de herramienta CASE que proporciona técnicas de validación mediante simulación es case/4/0 [36]. La herramienta permite modelar sistemas mediante diagramas entidad-relación, diagramas de estados y flujos de información, y validar estos últimos mediante su animación. Otros ejemplos son GRADE [80], un entorno de modelado de sistemas complejos que permite analizar el comportamiento descrito mediante simulación y prototipado, o SES/workbench [165], una herramienta de simulación centrada en el modelado de rendimiento y comportamiento que anima los modelos. Den-

tro de los entornos de modelado para UML, BridgePoint [32] incluye un verificador que anima modelos xtUML (*Executable and Translatable UML*) mediante una máquina virtual ejecutable. El verificador muestra la concurrencia del sistema animando sus máquinas de estados. La herramienta se puede integrar con otras de Eclipse.

Entre las herramientas que usan técnicas de análisis basadas en transformación, lo más habitual son los entornos de modelado que utilizan *model checkers* para realizar la verificación. Por ejemplo, la herramienta vUML [119] transforma modelos UML al lenguaje PROMELA, lenguaje de entrada del *model checker* SPIN [94] donde se realiza la verificación. En concreto, vUML permite verificar la consistencia de *statecharts* respecto a diagramas de colaboración. Detecta la presencia o ausencia de *deadlocks* y obtiene las invariantes que se preservan. En caso de detectarse un *deadlock* se genera un diagrama de secuencia que describe los pasos que llevan al mismo. Hugo [161] es otro proyecto que también usa SPIN para detectar *deadlocks* en diagramas UML. Al igual que en vUML, los modelos se transforman automáticamente a PROMELA, y si se detecta un error se genera un contraejemplo. Para realizar verificaciones más complejas hace falta conocer SPIN y la estructura de la transformación.

También existen entornos de modelado que se apoyan en herramientas de verificación no basadas en *model checking*. Por ejemplo, ACL2 [2] utiliza Lisp como lenguaje textual para especificar sistemas, lo que permite verificarlos mediante su ejecución o utilizando un probador de teoremas. Esto es posible ya que Lisp es tanto un lenguaje de programación como un lenguaje matemático. Sin embargo no es fácil construir modelos complejos con Lisp ni probar los teoremas (los autores apuntan que se necesitan meses para ser un usuario efectivo de ACL2). Otro ejemplo es FuZE [75], una extensión de un entorno de modelado orientado a objetos que genera de manera automática especificaciones Z a partir de modelos de análisis Fusion. El análisis se realiza invocando herramientas Z para verificar (con ZTC) o animar (con ZANS) la especificación Z resultante de la transformación. En el entorno de modelado de sistemas distribuidos PARAGON [169], los sistemas se traducen al álgebra ACSR para ser usadas por la herramienta de verificación XVERSA. XVERSA realiza análisis del espacio de estados para verificar propiedades de seguridad, comprobar la equivalencia de procesos y ejecutar el sistema. En este caso la verificación se realiza directamente sobre XVERSA, lo cual implica saber ACSR, conocer la herramienta de verificación y saber interpretar los resultados que ésta proporciona. Para terminar, un ejemplo de verificación UML es el proyecto OMEGA [147], donde los modelos se transforman a una representación interna denominada IF que es entrada de la herramienta usada para verificación, y las propiedades a verificar se definen bien mediante (una variante de) máquinas de estados o con restricciones temporales.

Respecto al lenguaje que todas estas herramientas de modelado utilizan para expresar las propiedades a verificar, algunas optan por usar el mismo lenguaje que usa la herramienta de verificación, otras usan diagramas de colaboración UML, y finalmente otras utilizan

alguna variante de máquinas de estados [147].

3.4.2. Herramientas de análisis basadas en *model checking*

Uno de los tipos de herramienta más usados para la verificación de sistemas (ya sea desde entornos de modelado o de modo autónomo) son las basadas en *model checking*. Aceptadas a nivel industrial, no requieren un gran esfuerzo ni conocimiento de las tecnologías subyacentes para poder utilizarlas. Como desventaja tienen el problema de la explosión de estados. A continuación se muestran las características de algunas herramientas de este tipo, de entre las numerosas existentes. En general, todas proporcionan un lenguaje para definir los sistemas y otro distinto para especificar las propiedades que se desea verificar.

Para empezar, una de las herramientas de *model checking* más utilizadas es SPIN (*Simple Promela INterpreter*) [94], la cual permite la simulación y verificación formal de sistemas software distribuidos. Los sistemas a verificar se deben expresar con un lenguaje nodeterminista denominado PROMELA (*PROcess MEta LAnguage*) que es bastante similar a C. Para definir las propiedades se pueden usar invariantes de procesos o sistemas (aserciones), lógica temporal lineal (LTL), autómatas Büchi[1] o propiedades "omega-regular". Si una propiedad es falsa genera un contraejemplo. SPIN tiene tres modos de funcionamiento: como simulador, como verificador exhaustivo para validar los requisitos de corrección especificados por el usuario, y como sistema de aproximación de pruebas para validar sistemas grandes. SPIN se ha usado para verificar sistemas muy diversos, como protocolos de comunicación de datos, sistemas operativos o algoritmos concurrentes. También hay entornos de modelado que la usan internamente para el análisis de modelos [119, 161].

COSPAN [7] es otro ejemplo de *model checker* que permite realizar especificaciones de sistemas distribuidos, analizar su comportamiento lógico o estocástico, y generar código C a partir de las especificaciones. El análisis lógico se lleva a cabo mediante verificación simbólica, y los algoritmos de análisis usan técnicas de reducción para tratar espacios de estados grandes. La extensión RTCOSPAN también permite modelar restricciones de tiempo real mediante componentes que incluyen los tiempos en que se pueden producir los eventos. De ese modo excluyen la computación de secuencias de comportamiento que son inconsistentes con la información temporal.

Un tercer ejemplo es NuSMV [40], una herramienta que implementa *model checking* simbólico basado en árboles de decisión binarios. En este caso los sistemas se deben especificar utilizando el lenguaje SMV o lógica proposicional, y pueden ser síncronos o asíncronos. Las propiedades a verificar se pueden expresar en lógica temporal CTL y LTL. Dispone de heurísticas para controlar la explosión de estados, e incluye análisis de invariantes, análisis de alcanzabilidad, computación de las características cuantitativas de un modelo, y gene-

[1] Un autómata de Büchi es un autómata definido sobre secuencias de entrada infinitas, en vez de sobre secuencias finitas.

ración de contraejemplos. Además de mecanismos de verificación, también permite simular un modelo para depurar su comportamiento.

También existen bastantes verificadores de sistemas hardware que utilizan Verilog o VHDL como lenguajes de entrada para especificar los sistemas [21, 31, 72]. Uno de ellos es FormalCheck [72], basado en COSPAN. FormalCheck proporciona tres algoritmos de verificación: enumeración de estados simbólica con diagramas de decisión binarios para modelos grandes, enumeración de estados explícita para modelos de menos de 1000 valores por estado, y auto-restringida para restringir la porción de diseño a verificar antes de realizar el análisis, acelerando de ese modo el proceso de búsqueda. Otro *model checker* que también usa Verilog como lenguaje de entrada es VIS (*Verification Interacting with Synthesis*) [31], aunque éste transforma internamente Verilog al lenguaje intermedio BLIF-MV. VIS utiliza CTL como lenguaje de propiedades, y realiza verificación de equivalencia de diseños, simulación basada en ciclos y síntesis jerárquica. Como último ejemplo, RuleBase [21] se diferencia de las dos herramientas anteriores en que usa un lenguaje de alto nivel llamado Sugar para especificar las propiedades, las cuales se transforman internamente a CTL. Además, en caso de que una propiedad no se verifique genera un diagrama temporal como contraejemplo y una explicación de cuál es el punto donde la propiedad deja de ser cierta por primera vez.

3.5. Medición y rediseño de software

Uno de los objetivos generales de la presente tesis es la generación de entornos visuales que integren mecanismos para medir y mejorar la calidad de los modelos. Como puede suponerse, tales mecanismos deberían ser generales para poder ser reutilizados o adaptados a cualquier LVDE para el que se quiera desarrollar un entorno. Por tanto esta sección recoge las propuestas de diversos autores para la definición de mecanismos de medición y rediseño genéricos, esto es, independientes del lenguaje.

La sección incluye un apartado que analiza distintos entornos de modelado que permiten la medición o rediseño de modelos, pero que además presentan características relevantes en el presente contexto (extensión de los mecanismos predefinidos o independencia del lenguaje).

3.5.1. Propuestas para la medición y rediseño genérico de software

Diversos autores proponen modelos que manejan conceptos abstractos para especificar medidas y rediseños genéricos, que posteriormente se pueden configurar para aplicarlos a lenguajes concretos. Estos modelos siguen la conocida premisa "*define una vez, reutiliza donde quieras*" para minimizar el esfuerzo a la hora de especificar medidas y rediseños para lenguajes nuevos. Lo más frecuente es que estas propuestas, aunque genéricas, se encuentren orientadas a un dominio específico particular (muy a menudo la orientación a objetos) y encaminadas más hacia la parte de implementación que hacia la parte de diseño. A continuación se estudian algunas de las propuestas más representativas.

Medición

El enfoque *Goal Question Metrics* (GQM) [120] está orientado al descubrimiento de patrones de medición reutilizables. Para ello se identifican los objetivos del proyecto software y se refinan a objetivos de caracterización más concretos, a partir de los cuales derivar los patrones de medición. Sin embargo, GQM no especifica cómo categorizar los patrones obtenidos ni reutilizarlos en distintos dominios.

Un enfoque distinto es el que siguen [135, 140]. En ambos casos los autores presentan enfoques basados en meta-modelado para especificar métricas genéricas para sistemas orientados a objetos. Para ello definen sendos meta-modelos que incluyen conceptos abstractos del dominio, como clase o atributo. Las métricas se definen de forma genérica usando los conceptos del meta-modelo. Posteriormente, para usar una de estas métricas sobre un lenguaje específico, basta con dar la relación existente entre los conceptos del lenguaje y los del meta-modelo. Nótese que como el cálculo de las métricas depende de

los conceptos del meta-modelo, no se puede aplicar a lenguajes que no incluyan los conceptos del mismo. Las métricas así definidas, por tanto, son independientes del lenguaje orientado a objetos pero dependientes del dominio, y excluyen su aplicación a lenguajes de otros dominios. Tampoco explotan el potencial de las métricas para actuar como detectores de malos olores que indiquen qué entidades son buenas candidatas para un rediseño. De estos dos enfoques, el propuesto en [135] permite además componer las métricas básicas predefinidas para calcular otras más complejas.

En [130] presentan un enfoque que también está basado en meta-modelado y es independiente del lenguaje. Parten de un meta-modelo con los conceptos del lenguaje, y las métricas se definen en un paquete separado que contiene una sola clase base llamada `Metric`. Definir una nueva métrica implica extender la clase base y declarar su función de medición mediante una consulta OCL parametrizada con los elementos apropiados del meta-modelo. El enfoque no permite componer métricas ni asociarles acciones.

El enfoque seguido en SPQR/20 [171] también es similar a los anteriores ya que proporciona una implementación de la función de medición, en este caso una versión extendida de puntos de función, que se puede aplicar a diversos lenguajes. Sin embargo no utiliza meta-modelado para formalizar los conceptos genéricos de los lenguajes, sino que su aplicación se realiza clasificando los lenguajes de acuerdo a niveles con características predefinidas.

También relacionado con la generación de métricas genéricas, existen diversos intentos para definir ontologías de medición software [77, 128]. Parte del trabajo desarrollado en la presente tesis, específicamente la parte concerniente a la definición de medidas que presenta el próximo capítulo, se basa en algunos de estos trabajos para definir un meta-modelo genérico para la definición de medidas.

Rediseño

Al igual que para medición, también existen algunas propuestas para la especificación de rediseños genéricos (o más frecuentemente de *refactorings* de código o modelos) que se pueden configurar para lenguajes concretos. Por ejemplo, [113] presenta este concepto junto con una implementación basada en programación estratégica en Haskell. El marco consiste en meta-programas que se pueden aplicar a distintos lenguajes de programación mediante paso de parámetros. Los *refactorings* así definidos son genéricos ya que se pueden aplicar a cualquier lenguaje. Sin embargo, especificar los parámetros apropiados en cada caso resulta complicado e implica conocer Haskell y la sintaxis abstracta del lenguaje específico. Además no provee mecanismos que faciliten la búsqueda del código candidato a reestructurar.

En [172] definen un LVDE para la evolución de modelos de dominio. Para ello, un modelo escrito en un LVDE se representa como un grafo, y la evolución de ese modelo cuando el lenguaje cambia se trata como una operación de reescritura que da como resultado un grafo conforme al lenguaje evolucionado. La transformación se especifica mediante una

secuencia de estructuras *Transform* que establecen la relación existente entre los objetos antes y después de la evolución. Dicha relación se define mediante un patrón de diagramas de objetos que relaciona las clases en los meta-modelos inicial y evolucionado. Las modificaciones pueden producir cambios en la semántica del lenguaje, y eliminar o reemplazar patrones sintácticos existentes.

3.5.2. Detección de posibilidades de rediseño

Existen diversas técnicas para guiar la aplicación de rediseños mediante la detección de lo que se conoce como "malos olores". En [103] detectan malos olores mediante el cálculo de invariantes que se infieren a partir de los programas. Cada invariante tiene asociado un *refactoring* e identifica dónde aplicarlo. Por ejemplo, una invariante sobre los parámetros de un método puede indicar si hay alguno innecesario porque no se usa, es constante o puede calcularse en función de otros. En total proponen seis posibles *refactorings* detectables mediante este tipo de análisis. En [180] los malos olores se identifican con meta-programación lógica. Para ello existe una serie de predicados lógicos qué evalúan qué partes del código son sensibles a error, y como resultado se aplica el conjunto apropiado de *refactorings* (de entre los predefinidos en una base de datos) para eliminar o reducir el error. Este marco es independiente del lenguaje usado.

Una aproximación distinta para identificar código o modelos candidatos a un rediseño son las métricas. En [167] usan este enfoque para sistemas orientados a objetos. Comparativamente, las métricas son una técnica bastante ligera cuando la búsqueda se realiza en sistemas grandes, y también más elegante para ciertos tipos de malos olores. Como desventaja está el hecho de que deben interpretarse, esto es, a veces el valor apropiado de una métrica puede depender del tipo de aplicación o comportamiento que queremos generar, de tal modo que distintos valores pueden ser apropiados o erróneos dependiendo del contexto.

3.5.3. Entornos de modelado con capacidad de medición y rediseño

La OMG ha reconocido recientemente la necesidad de herramientas para la modernización y evolución de software. Actualmente, su *Architecture-Driven Modernization Task Force* [3] está desarrollando un conjunto de estándares para el desarrollo de herramientas de modernización basadas en meta-datos que faciliten el análisis, visualización, *refactoring* y transformación de sistemas software existentes. Para ello ha publicado una solicitud de propuestas (*Request for Proposals*) [4] de paquetes de métricas y *refactorings* con el objetivo de construir un meta-modelo que los defina y permita su intercambio. Estos paquetes deberían ser lo suficientemente flexibles para adoptar nueva clases de métricas a un mínimo coste.

Aunque existen muchos entornos de modelado con capacidad de realizar mediciones, el conjunto de métricas que proporcionan suele estar predefinido en la herramienta, orientado a un dominio o lenguaje específico (el que soporta la herramienta) y las posibilidades de extensión son bastante limitadas. Existen algunas excepciones que cubren alguna de estas limitaciones por separado, aunque no todas ellas simultáneamente. Por ejemplo, la herramienta SDMetrics [164] permite definir métricas para modelos UML usando un lenguaje basado en XML (lo cual lo hace poco usable). ATHENA [181] también proporciona un conjunto de métricas base para lenguajes textuales que es extensible mediante la utilización de un LDE. Las métricas son independientes del lenguaje: proporcionando como entrada de la herramienta la especificación del lenguaje y los algoritmos de cálculo y evaluación de las métricas, éste genera un entorno de medición para el lenguaje correspondiente. Otra excepción es el entorno de reingeniería Moose [115], el cual implementa un motor para métricas software orientadas a objetos independientes del lenguaje (pero no del dominio). Proporciona más de treinta métricas predefinidas sin posibilidad de extensión, pero que pueden configurarse para cualquier lenguaje orientado a objetos relacionando los conceptos del lenguaje con los definidos en una representación independiente del lenguaje denominada FAMIX.

Por otro lado, la herramienta MetricView [137] es dependiente del lenguaje pero independiente de la métrica. Permite visualizar cualquier métrica referente a un elemento UML (y calculada en una herramienta externa) proyectándola sobre los diagramas en los que el elemento aparece. Para cada tipo de métrica el usuario sólo debe seleccionar cómo representarla y cuáles son sus valores extremos. Sin embargo no permite visualizar métricas para conjuntos de elementos (como se hace por ejemplo en [167]), y la interpretación de cada métrica y definición de valores extremos corre a cargo del usuario. Una versión posterior de la herramienta (MetricViewEvolution [137]) permite el cálculo de métricas dentro de la misma herramienta.

También existen algunas herramientas de modelado que permiten aplicar rediseños o *refactorings* sobre modelos, aunque este tipo de técnicas está más presente en entornos de programación que en entornos de modelado. Ya sea en uno u otro caso, los rediseños suelen estar orientados a un lenguaje específico, codificados a priori en la herramienta sin posibilidad de extensión, y la detección de las partes susceptibles al rediseño se tiene que hacer a mano. Algunos ejemplos son el *Refactoring Browser* [157] para código Smalltalk, o Together [179] para código Java y modelos UML. Una de las pocas herramientas que detecta automáticamente oportunidades de *refactoring* es SOUL [180]. Para ello utiliza un lenguaje basado en meta-programación lógica construido sobre el entorno VisualWorks Smalltalk que detecta automáticamente la existencia de malos olores y propone el conjunto de *refactorings* apropiados a aplicar en cada caso. De nuevo, esta herramienta es de dominio específico y el conjunto de malos olores detectables, así como el de *refactorings* propuestos, está predefinido y no se puede modificar.

Por último, como mostró el estudio de herramientas para generar entornos de modelado expuesto en la sección 3.2, no existe ninguna de ellas que proporcione soporte para incluir mecanismos de medición en los entornos generados. Para definir rediseños algunas permiten usar lenguajes de transformación de modelos pero, de nuevo, en ningún caso proporcionan soporte para guiar la ejecución de los rediseños en base a la detección de malos olores.

3.6. Desarrollo dirigido por modelos

Un paradigma de desarrollo reciente cuyo éxito depende en gran medida de la funcionalidad proporcionada por las herramientas de modelado usadas para su implementación es el Desarrollo de Software Dirigido por Modelos (DSDM) [188]. El DSDM se basa en el uso de modelos para especificar de manera eficiente la estructura y comportamiento de un sistema. Más aún, los modelos no sólo documentan el análisis y diseño del sistema, sino que constituyen su implementación y permiten la generación de código, casos de prueba y mecanismos de análisis de manera automática. Por tanto, una arquitectura basada en DSDM debe definir los conceptos y reglas disponibles para crear modelos, la notación usada para dibujarlos, qué elementos del mundo real representan sus elementos, y las transformaciones de los modelos a otros artefactos (incluyendo código) [13]. Con el término mega-modelo [67] nos referimos al modelo que especifica conceptos de DSDM y permite razonar sobre ellos.

La Arquitectura Dirigida por Modelos [131] (MDA) es una implementación de DSDM con estándares de la OMG (UML para el modelado de sistemas, MOF para el metamodelado, OCL para la especificación de restricciones y QVT para la transformación de modelos). En esta propuesta, los desarrolladores construyen modelos independientes de la plataforma (PIMs) utilizando lenguajes de alto nivel de abstracción. A continuación eligen la plataforma para la que quieren generar código, de tal modo que una transformación QVT se encarga de transformar los PIMs a modelos que incluyen aspectos de la plataforma (PSMs). Diferentes transformaciones QVT definidas en la arquitectura de desarrollo específica permiten automatizar las transformaciones para distintas plataformas. Por último, a partir de los PSMs se genera el código ejecutable de la aplicación. Adicionalmente, MDA contempla el modelado de requisitos del sistema y del modelo de negocio mediante modelos de mayor nivel de abstracción independientes de la computación (CIMs).

Como puede verse, la transformación de modelos es clave en el DSDM (y MDA) para refinar modelos entre distintos niveles de abstracción (transformación modelo-a-modelo de PIM a PSM) y generar código (transformación modelo-a-plataforma). También es posible encontrar transformaciones dentro del mismo nivel de abstracción, por ejemplo para implementar *refactorings* de modelos que mejoren alguna característica de calidad del diseño antes de la generación de código. La sección 3.3 recogía una selección de los lenguajes de transformación de modelos más utilizados en la actualidad. Idealmente, dentro del marco de DSDM, los lenguajes usados deberían proveer mecanismos para sincronizar los modelos origen y destino de una transformación frente a posibles cambios, esto es, sincronizar PIMs y PSMs así como PSMs y código.

3.6.1. Herramientas para el desarrollo dirigido por modelos

Actualmente el DSDM, y más en concreto MDA, están muy extendidos y cada día surgen nuevas herramientas que los soportan. En ocasiones estas herramientas están orientadas al desarrollo de aplicaciones en dominios concretos, a menudo la web por el tipo de tecnología que generan (EJB o Webservices). Este es el caso de herramientas como OptimalJ [149] o ArgoUML [11], siendo esta última la versión gratuita sobre la que se basan herramientas comerciales como Poseidon [153]. También existen otras herramientas que generan aplicaciones de ámbito general, como MagicDraw [124].

Sin entrar en detalle en el proceso de generación de código (por no ser una característica relevante para los propósitos de esta tesis), el enfoque más utilizado es el basado en plantillas. Una plantilla establece la relación entre el modelo de datos del sistema y el código a generar desde el mismo. Las plantillas incluyen referencias a las entidades que pertenecen al modelo de datos. Así, un motor de plantillas toma como entrada el modelo de datos de un sistema y las plantillas de generación de código, y reemplaza las referencias existentes en las plantillas por datos reales del modelo, obteniendo el código fuente. De algún modo las plantillas se pueden ver como un lenguaje de consultas que obtiene parte de la información de un modelo y cuyo resultado se muestra en un nivel distinto de abstracción (el código). Este enfoque lo utilizan herramientas como AndroMDA [9], OpenArchitectureWare [148] o StP/ACD [174]. A veces el uso de plantillas se complementa con patrones (OptimalJ). Los patrones dictan qué plantillas usar para la generación de código y cómo usarlas para implementar los componentes de la aplicación, dependiendo de los modelos definidos. Finalmente, otras herramientas como ArgoUML utilizan lenguajes procedimentales para la generación de código.

Sin embargo, más que en el proceso de generación de código, el presente estudio está interesado en cómo se realiza el modelado de sistemas en las herramientas, qué mecanismos para el aseguramiento de la calidad de los modelos de análisis y diseño proporcionan, y qué lenguajes de transformación modelo-a-modelo utilizan. Esa es la razón de que el apartado considere algunas herramientas CASE que, si bien no soportan todo el proceso de generación de código que una herramienta de DSDM requiere, complementan a otras herramientas que sí lo hacen. De hecho, la primera diferencia que podemos encontrar entre herramientas para DSDM si atendemos a sus capacidades de modelado es si delegan esta tarea a una herramienta CASE externa (como ocurre con AndroMDA) o si integran editores de modelos para ello (como ArgoUML, Objecteering [146], OptimalJ). El problema con el primer tipo de herramientas es que exige cambiar de herramienta según la actividad realizada en cada momento.

Respecto a los lenguajes utilizados para el modelado de sistemas, existen herramientas que se ciñen a la especificación MDA y por tanto utilizan UML (ArgoUML, MagicDraw, Objecteering, OptimalJ), otras utilizan lenguajes basados en UML (Fujaba), y finalmen-

te hay algunas que permiten definir o usar LDEs o de propósito general (AndroMDA, StP/SE [174]).

Por lo general, las herramientas que no integran editores de modelos tampoco proporcionan mecanismos de consistencia entre modelos (aunque sí propagación de cambios de modelos a código), ni de rediseño o medición de modelos. Estas tareas se dejan a los editores de modelos, o bien se realizan sobre el código generado (proceso que resulta más costoso). Entre las herramientas que integran editores de modelos, algunas se encargan de verificar la consistencia entre modelos (ArgoUML, MagicDraw, Objecteering), realizan *refactorings* de modelos o guían hacia la consecución de buenas prácticas y patrones de diseño (ArgoUML, MagicDraw, OptimalJ), o generan informes con propiedades o métricas del sistema en desarrollo (MagicDraw, Objecteering). Como las herramientas manejan un lenguaje fijo (UML en su mayoría), los mecanismos para realizar las verificaciones y análisis están codificados a priori en la herramienta y son adecuados para el dominio del lenguaje utilizado.

Respecto a las transformaciones modelo-a-modelo, los lenguajes utilizados dependen de la herramienta determinada. Por ejemplo, algunas herramientas utilizan ATL (como por ejemplo AndroMDA y OpenArchitectureWare), transformación de grafos o grafos triples (Fujaba), patrones (OptimalJ) o lenguajes procedimentales (ArgoUML).

Del estudio realizado se concluye que de las muchas herramientas de modelado existentes para DSDM, una gran mayoría sigue el enfoque defendido en la sección 2.4 de integrar mecanismos para verificar la corrección de los modelos y garantizar la calidad de los sistemas desde las fases iniciales del desarrollo, cuando el coste de aplicar tales mecanismos y resolver los problemas detectados es menor [11, 124, 146, 149]. Además, en las herramientas analizadas, tanto los mecanismos de verificación como los lenguajes origen y destino de las transformaciones suelen estar fijos (por ejemplo, generación de código Java desde modelos UML). El objetivo de esta tesis es ser capaces de generar entornos visuales de modelado adaptados a las necesidades de mercado actuales. Para ello la generación de meras herramientas de dibujo no es suficiente, sino que éstas deben integrar mecanismos para la consistencia sintáctica y semántica entre modelos, verificación, análisis, medición y que en general ayuden a mejorar la calidad de los modelos (y en consecuencia del sistema resultante que representan). Más aún, es necesario poder adaptar los mecanismos de verificación al LVDE que se vaya a utilizar, dando los resultados del análisis en el contexto del lenguaje original.

3.7. Conclusiones

Este capítulo ha revisado el estado del arte de los entornos de modelado (CASE) actuales, centrándose en aquellas características que ayudan al diseñador a mejorar algún aspecto de la calidad de sus modelos. En concreto se ha estudiado qué mecanismos proporcionan para verificar o mantener la consistencia entre modelos en entornos multi-vista, cómo se realiza el análisis de los modelos de diseño y qué propiedades consiguen demostrar, o si permiten la animación de modelos, la obtención de métricas de calidad o la ejecución de *refactorings* de modelos. Del estudio se puede concluir que es una tendencia habitual enriquecer las herramientas de modelado con tales mecanismos. De hecho, en paradigmas de desarrollo dirigidos por modelos es crucial disponer de herramientas que integren tales aspectos para el control de la calidad de los modelos.

El capítulo también ha explicado en qué consisten los dos enfoques principales para la generación de entornos de modelado, que son el meta-modelado y la transformación de grafos. Hay dos hechos que parecen inclinar la balanza hacia la utilización del primero frente al segundo: que varias herramientas basadas en transformación de grafos han parado su desarrollo para pasarse al meta-modelado, y que los editores de entornos visuales más recientes siguen este paradigma. El hecho de que la definición del lenguaje sea un modelo (el meta-modelo) hace el enfoque más homogéneo. Con transformación de grafos hay que manejar dos conceptos: la gramática del lenguaje y los modelos. En cambio, los meta-modelos son modelos y por tanto se pueden analizar y manipular como cualquier otro modelo. La OMG y la especificación de UML han impulsado mucho el uso de meta-modelos.

Por otro lado, como se deriva del estudio de herramientas para el DSDM, el mercado actual necesita entornos de modelado completos y ricos (esto es, con funcionalidades añadidas). Por tanto, las herramientas que automatizan su generación deberían ser capaces de crearlos. Sin embargo, la mayoría de entornos de meta-modelado y editores de entornos visuales generan herramientas que permiten básicamente la especificación de modelos, lo que resulta insuficiente. Algunas como GME, MetaEdit+ o Pounamu van más allá proporcionando mecanismos de consistencia entre modelos, aunque tales mecanismos se generan en algún lenguaje de programación y resultan complicados (cuando no imposible) de modificar. Funcionalidades adicionales en los entornos generados, tales como la integración de métodos de verificación, medición o rediseño en los entornos generados se tiene que codificar a mano o, en el mejor de los casos, mediante algún lenguaje de transformación de modelos. En otros casos las herramientas generadas permiten exportar los modelos a XMI para su integración con herramientas de análisis, dejando al usuario final del entorno las tareas de integración y análisis de resultados.

Algunas herramientas como OpenArchitectureWare (que sin embargo no proporciona soporte para LVDEs) están siguiendo esta tendencia de generar entornos visuales avanzados, para lo que integran un conjunto de herramientas adicionales que ayudan a tareas

comunes en el desarrollo dirigido por modelos como la generación de código, la transformación de modelos o la generación de informes. El hecho de que algunas de esas herramientas estén integradas en Eclipse puede facilitar la interoperabilidad con otras herramientas. Sin embargo, todas esas herramientas relacionadas tendrían que estar configuradas (probablemente usando el meta-modelo del LVDE como núcleo de tal configuración) y fuertemente integradas para el dominio dado.

El capítulo también ha mostrado una selección de lenguajes de transformación de modelos, clasificados según si son textuales o visuales, declarativos o imperativos, y formales o semi-formales. La elección del lenguaje más apropiado depende del tipo de problema que se quiere resolver en cada caso. Para los objetivos de esta tesis se desea utilizar técnicas de transformación formales que permitan derivar propiedades tanto de las transformaciones (confluencia, terminación, ...) como de los modelos resultantes de la transformación (equivalencia semántica, corrección sintáctica, ...). Además se prefiere el uso de técnicas visuales siempre que sea posible por resultar más intuitivas, al menos para usuarios de entornos visuales de modelado. Por último, se necesita definir tanto manipulaciones de modelos como transformaciones modelo-a-modelo, y que además estas últimas creen relaciones entre los modelos origen y destino de la transformación para poder expresar conceptos de uno en términos del otro. Por todas estas razones, para la resolución de esta tesis se ha elegido la transformación de grafos como lenguaje formal para transformaciones *in-place* y la transformación de grafos triples para transformaciones modelo-a-modelo, ya que ambos cumplen las propiedades deseadas de ser formales y visuales.

Capítulo 4

Marco de especificación, análisis y generación de entornos para lenguajes visuales de dominio específico

Los LVDEs se usan a menudo durante las fases de análisis y diseño de un sistema para capturar su comportamiento y estructura en forma de modelos. Como esos modelos se usan de guía para construir el sistema, las características del producto final (no sólo su corrección, sino también su facilidad de mantenimiento, usabilidad, etc.) dependen de ellos, especialmente en paradigmas de desarrollo dirigido por modelos. Por tanto, garantizar la calidad de estos modelos resulta clave para el éxito del sistema resultante. Además, existen otras razones que refuerzan la idea de incorporar el control de la calidad desde el inicio de un desarrollo software, como por ejemplo el hecho de que la verificación de un sistema sea menos costosa que la del sistema en sí, o que resulte más fácil solucionar un error de diseño al comenzar un proyecto que cuando éste avanza en el tiempo [23, 154].

Existen diversos mecanismos para gestionar la calidad en un proyecto software, como por ejemplo el uso de procedimientos, estándares o procesos. Dentro de la fase de modelado del software, una forma de introducir estos y otros mecanismos en la práctica habitual es disponer de herramientas integradas en los entornos de modelado utilizados por diseñadores y desarrolladores, las cuales permitan la validación, verificación, medición y rediseño de modelos, así como probar su conformidad respecto a estándares. Estas herramientas deberían estar adaptadas al dominio de aplicación y al perfil del usuario que las van a utilizar, o de otro modo su potencial podría quedar desaprovechado. El problema es que desarrollar tales herramientas para la gran cantidad de notaciones y entornos de modelado existentes es bastante costoso y requiere conocer el lenguaje(s) de programación en que se construirán. Por añadidura, estas herramientas tienden a desarrollarse desde cero, sin beneficiarse de desarrollos previos (a pesar de que muchos de estos mecanismos son comunes para distintas áreas de conocimiento), lo que lo convierte en una tarea repetitiva. Por estas razones es frecuente que los entornos visuales que se implementan no incorporen mecanismos para controlar la calidad de los modelos, funcionalidad que resulta deseable para garantizar la calidad de los productos generados a partir de ellos. En otros casos, los entornos visuales proporcionan mecanismos de medición y rediseño generales [96, 125, 157, 164], pero no se le dice al desarrollador cómo interpretar las mediciones o cuándo aplicar los rediseños en

dominios concretos, por lo que parte de su potencial se desaprovecha.

En este capítulo se presenta un conjunto de soluciones para la especificación y generación de entornos visuales para LVDEs que integran mecanismos para el control de la calidad (necesidades expresadas e implícitas) de los modelos construidos en tales entornos. Los entornos, junto con los mecanismos de control de calidad, se generan principalmente a partir de descripciones gráficas de alto nivel para facilitar la labor del desarrollador del entorno visual. El capítulo está dividido en dos grandes secciones. En la primera de ellas se presenta un marco para la definición de LVDEs multi-vista con soporte para validar la consistencia inter- e intra- diagrama y el análisis de modelos, con el objetivo de integrar mecanismos formales de verificación de modelos en entornos visuales. En la segunda sección se presenta un marco para la especificación visual de medidas y rediseños para LVDEs con el objetivo de integrar mecanismos para la medición y mejora de la calidad en entornos visuales. Ambos enfoques han sido implementados e integrados en la herramienta de meta-modelado AToM^3 [51], por lo que se incluyen sendos apartados sobre las herramientas generadas.

4.1. Soporte para lenguajes visuales de dominio específico multi-vista

Un LVDE multi-vista está formado por una familia de puntos de vista (o tipos de diagrama) distintos aunque relacionados, cada uno de ellos diseñado para la especificación de un aspecto particular del sistema (estructura, comportamiento, etc.). Al especificar un sistema utilizando un lenguaje de este tipo, existen distintos tipos de construcciones (agrupadas bajo el nombre genérico de *vista*) que se utilizan con distintos objetivos y se construyen de manera distinta. En primer lugar, el usuario define el sistema mediante un conjunto de *vistas del sistema* conforme a los puntos de vista que proporciona el lenguaje. También puede realizar consultas sobre una o varias vistas del sistema que dan como resultando una *vista derivada* del mismo. Si las consultas vienen predefinidas junto con la definición del lenguaje multi-vista se obtienen *vistas orientadas a audiencia* que muestran una parte de la información del sistema para un determinado tipo de usuario. Finalmente, las *vistas semánticas* de un sistema contienen todo (o parte de) el sistema expresado en un dominio semántico que proporciona técnicas adecuadas para su análisis. Estas últimas no son visibles al usuario, sino que se utilizan como herramienta interna de análisis del sistema y el resultado se muestra al usuario en la notación del lenguaje multi-vista. La figura 4.1 muestra un diagrama que refleja esta situación, en la que un usuario utiliza un LVDE multi-vista para definir el modelo de un sistema.

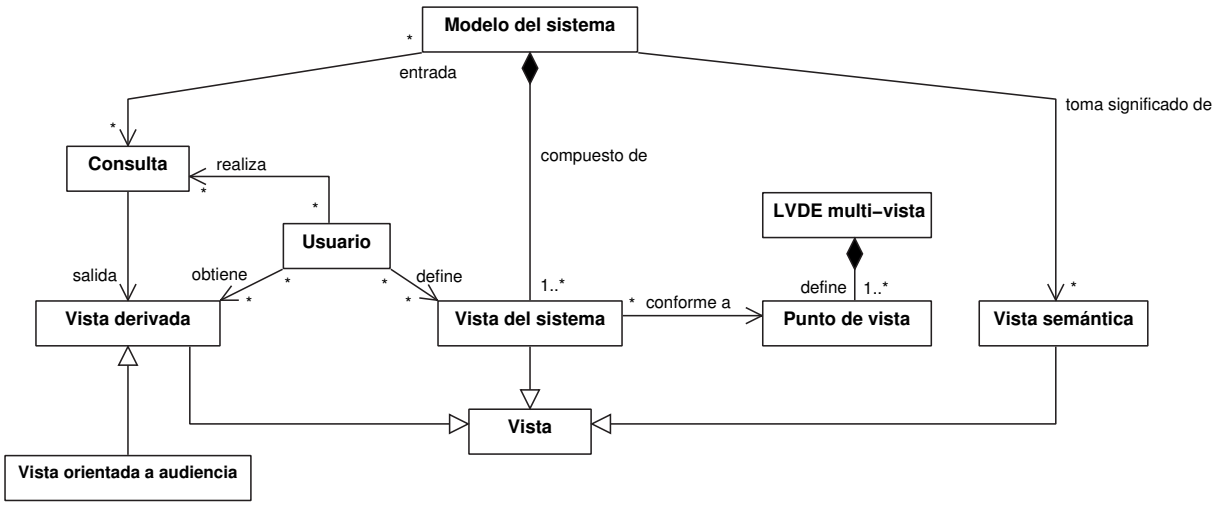


Figura 4.1: *Diagrama conceptual del uso de un LVDE multi-vista para definir un sistema*

En esta sección se presenta un marco gráfico y formal para la especificación de los distintos tipos de vista que, además, proporciona un conjunto de técnicas y herramientas para la definición de mecanismos de consistencia entre ellas. El objetivo es ayudar a la obten-

ción de sistemas que cumplan los requisitos explícitos de calidad (corrección, compleción y fiabilidad). Para ello se basa en el meta-modelado para describir la sintaxis de los LVDEs multi-vista, y en sistemas de transformación de grafos triples para:

1. Sincronizar y mantener la consistencia entre las vistas de un sistema mediante la construcción de un repositorio común que permita la propagación de cambios.

2. Generar vistas derivadas y dirigidas a audiencia como resultado de una consulta a un modelo base (ya sea una vista del sistema o el repositorio), así como para mantenerlas sincronizadas respecto a cambios en el modelo base. Como lenguaje de consulta se utilizan patrones visuales, a partir de los cuales se derivan los sistemas de transformación de grafos adecuados en cada caso.

3. Generar vistas semánticas a partir de una vista del sistema o del repositorio. El resultado obtenido tras el análisis de la vista semántica se anota en el modelo inicial mediante el uso de patrones triples.

La sección contiene cinco apartados. El primero comienza esbozando las extensiones realizadas a los sistemas de transformación de grafos triples definidos en [162] para permitir su integración en el marco multi-vista presentado. Los detalles de la formalización de tales extensiones se pueden consultar en el apéndice C. A continuación los tres siguientes apartados muestran el marco para la definición de vistas del sistema, derivadas y semánticas respectivamente. Finalmente, el último apartado muestra la implementación de un prototipo para la especificación de LVDEs multi-vista y posterior generación de entornos visuales para ellos.

4.1.1. Sistemas de transformación de grafos triples

Andy Schürr propuso las gramáticas de grafos triples como mecanismo de alto nivel para la sincronización de dos modelos relacionados a través de un grafo correspondencia, estructura a la que denominó *grafo triple* [162]. Tal como explicó la sección 2.2, los grafos de un grafo triple son grafos etiquetados atribuidos, y existe una función desde los nodos del grafo correspondencia a los nodos de los otros dos grafos. Además, las reglas de una gramática triple son monotónicas (esto es, permiten la creación pero no el borrado de elementos).

El marco multi-vista que presenta los siguientes apartados se basa en el uso de gramáticas de grafos triples para la sincronización de modelos y para la transformación inter-formalismo. La razón es que, como explicaba la sección 3.3, este lenguaje de transformación tiene un gran potencial para expresar transformaciones modelo-a-modelo, puesto que un grafo triple permite mantener limpiamente separados los modelos origen y destino de la transformación, y relacionarlos mediante correspondencias entre sus elementos. Además, su base formal proporciona técnicas para demostrar propiedades de la transformación, como su terminación o confluencia. Sin embargo, se ha necesitado extender el concepto de grafo triple para permitir la definición de grafos más complejos y similares a los utilizados en problemas de modelado reales (donde, por ejemplo, es habitual que los grafos tengan un tipo), así como para flexibilizar las relaciones entre los grafos del grafo triple. En concreto, las extensiones realizadas a la definición de grafo triple incluyen:

- la extensión del codominio de la función que va desde los nodos del grafo correspondencia a los nodos de los otros dos grafos, para permitir morfismos no sólo a nodos sino también a aristas o dejarlos indefinidos.

- la asignación de un tipo a los grafos triples mediante la definición de un grafo triple de tipos (similar a un meta-modelo) que puede contener relaciones de herencia en nodos y relaciones.

También se ha necesitado extender la definición de gramática de grafo triple para permitir reglas no monotónicas, incluir condiciones de aplicación, y que puedan ser utilizadas para transformaciones de modelos genéricas (en vez de sólo para la especificación de reglas operacionales). Por este motivo hablaremos de Sistemas de Transformación de Grafos Triples (TGTSs[1]). La definición de regla de un TGTS y su derivación en un grafo se ha formalizado mediante el enfoque algebraico *Double Pushout* (DPO) [61]. Los detalles de la formalización están disponibles en el apéndice C. A continuación se presenta un resumen con los principales conceptos.

[1]Como norma general, los acrónimos de términos técnicos se utilizarán en su forma inglesa para facilitar la lectura de la literatura relacionada

La definición formal de grafo triple se basa en el concepto de $E-grafo$ [61], el cual extiende la noción de grafo regular para permitir la atribución de nodos y aristas. Los valores que pueden asignarse a los atributos se representan mediante nodos de datos, y los atributos se representan mediante aristas que conectan los nodos y aristas del grafo con nodos de datos. Un $TriE-grafo$ está formado por tres E-grafos (denominados origen, destino y correspondencia) y dos funciones de correspondencia que van desde los nodos del E-grafo correspondencia a los nodos y aristas de los otros dos E-grafos. Las funciones de correspondencia también pueden estar indefinidas para algún valor en el dominio, lo que se modela asignando como imagen un elemento especial en el codominio denominado "$\cdot$". Una función entre dos TriE-grafos está formada por tres morfismos de grafos (uno por cada E-grafo del TriE-grafo) más un conjunto de restricciones adicionales que preservan las funciones de correspondencia. El conjunto de objetos TriE-grafo junto con los morfismos entre TriE-grafos forman la categoría **TriEGrafo**, donde los primeros son los objetos y los últimos las flechas. Con el objetivo de estructurar los posibles valores de atributos en tipos y de definir operaciones sobre ellos, los TriE-grafos se complementan con un álgebra sobre una signatura apropiada, dando lugar a la categoría **TriAGrafo**. Finalmente, a los TriAGrafos se les proporciona un tipo mediante un grafo triple de tipos (similar a un meta-modelo) que puede contener relaciones de herencia entre nodos y aristas, más una función de TriAGrafo que va desde el grafo al grafo de tipos. De este modo es posible definir grafos triples con tipos y atributos (abreviado ATT-grafos), y modelarlos como objetos en la categoría *slice* **TriAGrafo/TriATG**, también escrito $\mathbf{TriAGrafo}_{TriATG}$.

La figura 4.2 muestra a la izquierda un ATT-grafo sobre el grafo triple de tipos de su derecha. Para facilitar su legibilidad se muestra con una notación similar a UML, donde los nodos y aristas están etiquetados con su tipo, y los atributos aparecen en una caja seguidos de su valor. Por lo general esa será la notación utilizada en el presente capítulo, excepto en el apartado 4.1.5 donde se utilizará la sintaxis concreta del LVDE. Los tres grafos del ATT-grafo están separados por líneas discontinuas, y corresponden a una máquina de estados (origen), una red de Petri coloreada (destino) y un grafo correspondencia. Las funciones de correspondencia se representan mediante flechas discontinuas desde los nodos del grafo correspondencia a los nodos y aristas de los otros dos grafos. En el ejemplo, los estados de la máquina de estados están relacionados con los lugares de la red de Petri coloreada, así como las transiciones de uno y otro grafo (nótese que en ese caso la relación se establece entre una arista y un nodo). Si una función de correspondencia está indefinida (lo cual no ocurre en el ejemplo) se representaría gráficamente como un nodo del grafo correspondencia desde el cual no sale alguna de esas flechas. Igualmente es posible tener elementos en los grafos origen y destino que no pertenecen al dominio de la función de correspondencia, tal y como ocurre con los arcos de entrada y salida de la red de Petri o con el lugar llamado `Input`.

Para manipular ATT-grafos utilizamos reglas DPO que contienen ATT-grafos en sus

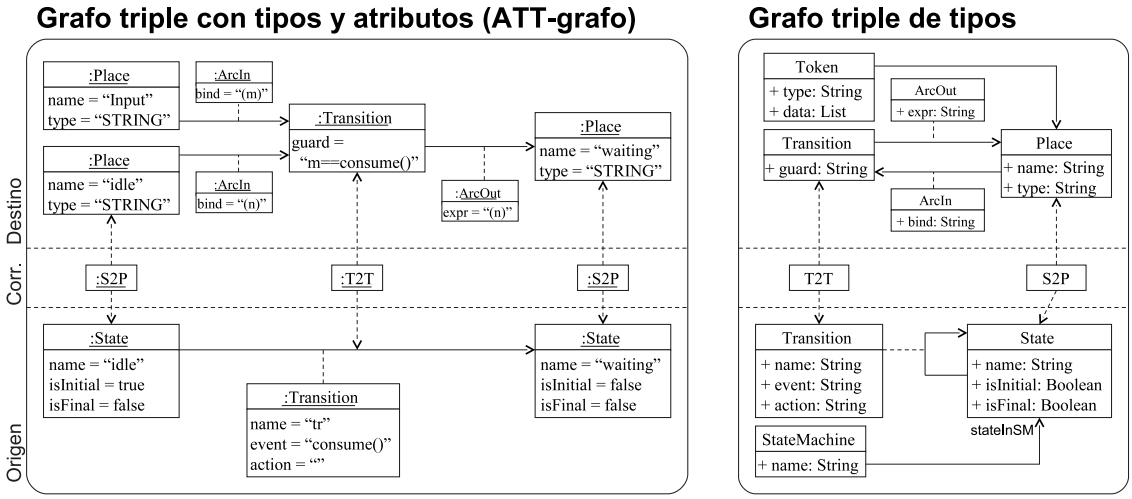


Figura 4.2: *Ejemplo de ATT-grafo y grafo triple de tipos*

partes derecha e izquierda, así como en las premisas y consecuencias de sus condiciones de aplicación (véase sección 2.2 para una definición precisa de regla y derivación DPO). Originalmente el enfoque DPO se definió únicamente para la transformación de grafos. Sin embargo recientemente Ehrig et al. [61] han extendido la teoría para permitir la transformación de objetos de cualquier categoría HLR adhesiva (débil) [110]. En el apéndice C se demuestra que la categoría $\mathbf{TriAGrafo}_{TriATG}$ es una categoría de este tipo. En consecuencia, podemos usar el enfoque DPO para transformar ATT-grafos.

La figura 4.3 muestra una regla triple que crea un nodo de tipo *Place* en el grafo destino y lo relaciona con un nodo existente de tipo *State* en el grafo origen a través de un nodo de correspondencia. La NAC no permite aplicar la regla si el estado ya está relacionado con otro lugar. La regla se muestra usando la notación que introdujo el apartado 2.2.1: el componente K con los elementos comunes en LHS y RHS (en este caso el nodo "1") no se muestra, y sólo los elementos que pertenecen a K (que son los que se preservan) aparecen numerados.

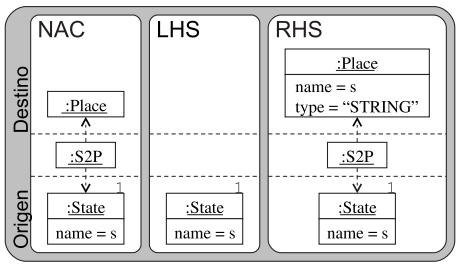


Figura 4.3: *Ejemplo de regla de grafo triple con tipos y atributos*

Técnicas de análisis de sistemas de transformación de grafos triples

La ventaja de proporcionar una definición formal para los TGTSs (basada en DPO) es que permite aplicar los resultados obtenidos en la teoría clásica de transformación de grafos sobre ellos. En el próximo apartado se usarán algunos de esos resultados para demostrar la terminación y confluencia de ciertos TGTSs. Idealmente, si existiese una herramienta que permitiera "parametrizar" la categoría sobre la que se trabaja, podríamos usarla con grafos triples. Sin embargo, como no la hay, se usará la herramienta AGG [5] que presentó la sección 3.3 como soporte para estudiar la confluencia, ya que es la única herramienta que automatiza el análisis de pares críticos para transformación de grafos. A continuación se muestra cómo usar AGG para representar grafos triples y TGTSs, y se presenta un ejemplo que ilustra cómo interpretar el resultado de los análisis que AGG proporciona.

En primer lugar, los grafos que maneja AGG son grafos tipados atribuidos. La definición de un grafo triple de tipos se puede modelar en AGG como un grafo normal de tipos, donde las funciones de correspondencia se definen mediante relaciones que tienen origen en los nodos de correspondencia y cardinalidad 0..1 (esto es, la función puede estar definida o indefinida). Además, como el destino de una relación nunca puede ser otra relación, cualquier enlace que sea destino de una función de correspondencia se debe modelar como un nodo. Dicho nodo tendrá obligatoriamente una relación con los nodos origen y destino de la relación que representa, de tal modo que sólo pueda existir si también existen los dos nodos que relaciona.

Por ejemplo, la figura 4.4 muestra la especificación en AGG del grafo de tipos equivalente al grafo triple de tipos que muestra la figura 4.2. Las funciones de correspondencia se muestran con línea discontinua para diferenciarlas del resto, pero son relaciones iguales a las demás. Obsérvese que como la relación `Transition` era destino de una función de correspondencia en el grafo triple de tipos, en AGG se modela mediante un nodo.

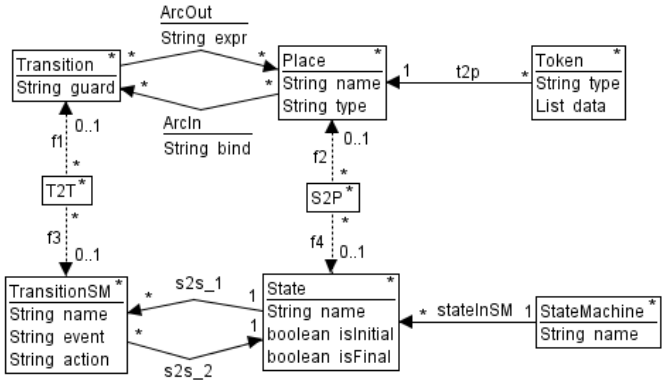


Figura 4.4: *Especificación del grafo triple de tipos de la figura 4.2 en AGG*

Para especificar un ATT-grafo conforme a un grafo triple de tipos en AGG se debe

definir un grafo normal tipado sobre el grafo de tipos que equivale al triple. De este modo, definir un TGTS implica definir reglas de transformación donde los ATT-grafos de las partes izquierda, derecha y condiciones de aplicación de las reglas triples se modelan como grafos normales con el tipo explicado. Además, si la parte izquierda de una regla contiene una función de correspondencia indefinida, hay que añadir una NAC a la regla que especifique explícitamente su inexistencia.

A modo de ejemplo, la figura 4.5 muestra la especificación de dos reglas triples usando AGG. El grafo de tipos de los componentes de cada regla es el que muestra la figura 4.2. La primera regla (que corresponde a la especificación de la regla triple de la figura 4.3) crea un objeto *place* conectado a un estado si éste no está ya conectado a un objeto de este tipo (NAC). La segunda regla es similar pero creando una transición de red de Petri asociada a una transición de máquina de estados.

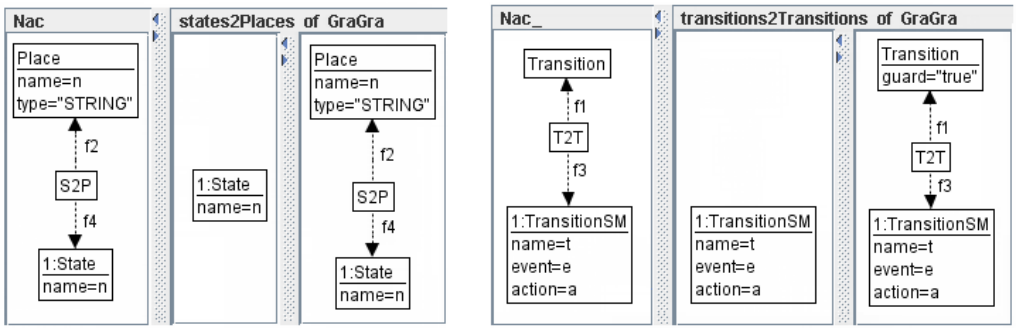
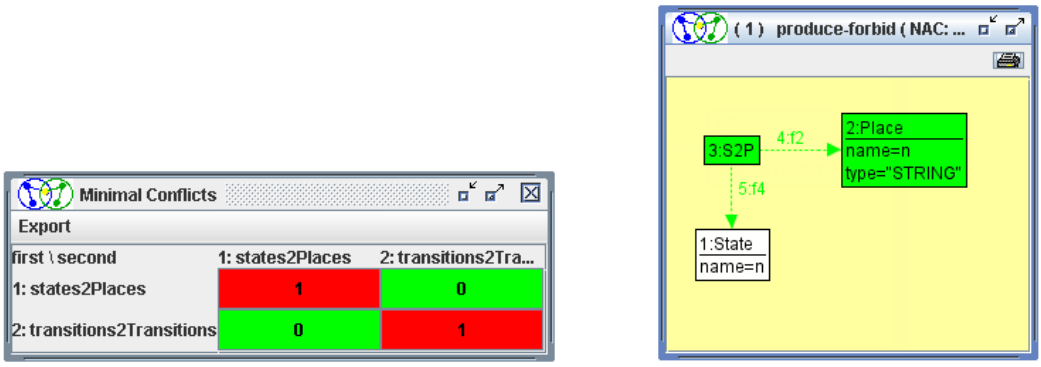


Figura 4.5: *Especificación de dos reglas triples en AGG*

Una vez definido un TGTS de la manera expuesta, es posible usar las técnicas de análisis de pares críticos que AGG proporciona como base para estudiar su confluencia. Este análisis obtiene todas las posibles aplicaciones de dos reglas que pueden ocasionar un conflicto, esto es, todos los grafos mínimos tales que aplicar una de las reglas imposibilita aplicar la otra. Esto puede ocurrir cuando la primera regla modifica atributos o borra elementos que la segunda regla necesita para su aplicación, o bien si crea nuevos elementos que son iguales a una NAC de la segunda.

La figura 4.6(a) muestra el resultado de efectuar el análisis de pares críticos sobre las dos reglas anteriores. El resultado es una matriz que tiene las reglas de la gramática como filas y como columnas. Cada casilla (i, j) indica el número de conflictos que pueden surgir de aplicar la regla de la fila i y a continuación la de la columna j. En este caso vemos que siempre se puede aplicar una de las reglas y luego la otra ya que no hay conflictos entre ellas, lo cual significa que el resultado de aplicarlas es confluente. Sin embargo, aplicar una regla puede impedir su nueva aplicación (esto es, cada regla es excluyente consigo misma). Nótese que eso es precisamente lo que se pretendía en el ejemplo al definir las NACs. De hecho, provocar un conflicto de una regla consigo misma es una técnica muy



(a) Matriz de conflictos (b) Grafo conflicto

Figura 4.6: *Análisis de pares críticos en AGG*

común para asegurar que una transformación termina. AGG permite ver el grafo mínimo a partir del cual surge el conflicto. En la figura 4.6(b) se muestra este grafo para el caso de la primera regla consigo misma. La primera aplicación de la regla crea los nodos coloreados, y efectivamente se comprueba que la regla no se puede volver a aplicar porque su NAC lo prohíbe (de ahí el nombre del par crítico: *produce-forbid*). Con la otra regla ocurre lo mismo.

Para demostrar la confluencia de una gramática o TGTS hay que demostrar la confluencia de sus pares críticos (pares de reglas con conflictos detectados por AGG). En el ejemplo, los pares críticos son confluentes porque aplicar dos veces la misma regla lleva al mismo resultado, ya que ha de aplicarse en subgrafos independientes. Como todos los pares críticos del TGTS confluyen se puede concluir que el TGTS es confluente.

4.1.2. Vistas del sistema

Los LVDEs multi-vista están formados por un conjunto de tipos de diagrama distintos. Cada uno tiene su propio meta-modelo que captura un aspecto o punto de vista del sistema. Sin embargo, todas esas definiciones se basan en un único meta-modelo que relaciona los conceptos de sus sintaxis abstractas (tal es el caso, por ejemplo, en la definición de UML [182]). Los distintos puntos de vista se pueden solapar en el meta-modelo único, lo que significaría que el mismo concepto aparece en puntos de vista distintos del lenguaje. Este hecho puede ocasionar inconsistencias al especificar las vistas de un sistema utilizando el LVDE multi-vista, ya que si un elemento aparece en más de un tipo de diagrama, aumenta la posibilidad de que en alguno de ellos la información no sea coherente con el resto (debido a un error en la especificación, o a la modificación de uno de ellos pero no del resto). Esta situación también puede surgir si el mismo elemento aparece en vistas del mismo tipo ya que un punto de vista se solapa plenamente consigo mismo. Por tanto resulta clave proporcionar mecanismos que se encarguen de garantizar la consistencia sintáctica entre las vistas del sistema, e identificar qué situaciones pueden dar lugar a una inconsistencia (esto es, qué elementos comparten cada dos puntos de vista).

Esta situación se ilustra en la figura 4.7 para un pequeño subconjunto de UML[2] [182] que define clases, asociaciones binarias, relaciones de herencia entre clases, máquinas de estados, estados y transiciones. El meta-modelo mezcla información estática y dinámica del sistema, razón por la cual se definen dos puntos de vista distintos sobre el lenguaje: el primero incluye sólo la información estructural (diagrama de clases) y el segundo sólo el comportamiento (máquina de estados). De hecho, ya que en el segundo punto de vista sólo interesa conocer la identidad de la clase a la que se asocia la máquina de estados, ésta sólo tiene su nombre como atributo. La intersección de los dos puntos de vista es la clase. Por tanto, si una clase cambia en un diagrama de clases, los cambios deberán reflejarse en los demás diagramas de clases que la contienen (ya que la intersección de un punto de vista consigo mismo es igual al punto de vista) y en las máquinas de estado. En cambio, si cambia una asociación, no es necesario propagar los cambios a las máquinas de estados ya que la asociación no pertenece a la intersección de los dos puntos de vista.

La figura 4.8 formaliza la definición de un LVDE multi-vista utilizando teoría de categorías (véase el apéndice A para una introducción a teoría de categorías). Como se muestra a la izquierda, un LVDE multi-vista está definido mediante un grafo de tipos con atributos TG^{MV}, que es su meta-modelo. Los distintos puntos de vista TG^{VP} están incluidos en él, aunque en un enfoque más general podría ser cualquier función. Por cada pareja de puntos de vista TG^{VP_i} y TG^{VP_j}, la parte que solapa $I^{VP_{i,j}}$ se calcula como el *pullback* de los respectivos grafos de tipos y TG^{MV}, lo que hace que el cuadrado conmute (es decir,

[2]Aunque UML no es un LVDE se ha decidido usarlo como ejemplo a lo largo de este capítulo ya que, al ser una notación ampliamente conocida, facilitará la comprensión del enfoque propuesto.

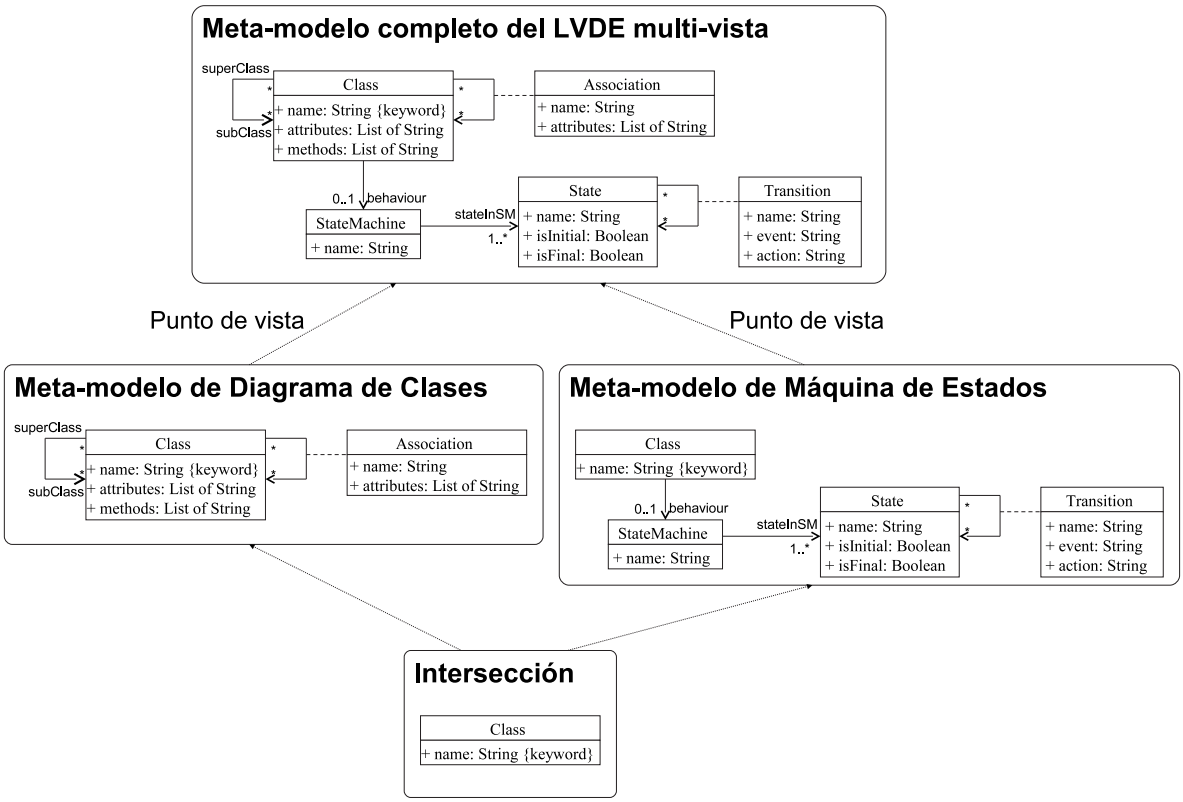


Figura 4.7: *Definición de un subconjunto de UML*

$id_i \circ o_{i,j} = id_j \circ o_{j,i}$). Nótese que $I^{VP_{i,i}} = TG^{VP_i}$. En el caso más habitual, todo elemento de TG^{MV} se proyecta en algún punto de vista TG^{VP}. Si esto es así, TG^{MV} es el colímite de todos los puntos de vista y su intersección. A nivel de modelo (derecha), el usuario construye vistas del sistema conforme a algún punto de vista (es decir, existe una función de tipado desde cada vista G del sistema a algún punto de vista), pudiendo definir más de una vista del mismo punto de vista. Para garantizar la consistencia sintáctica de las vistas se construye un modelo *repositorio* que las amalgama. El repositorio es el colímite de todas las vistas y su intersección, y existe una función de tipado desde el mismo al grafo de tipos del LVDE multi-vista.

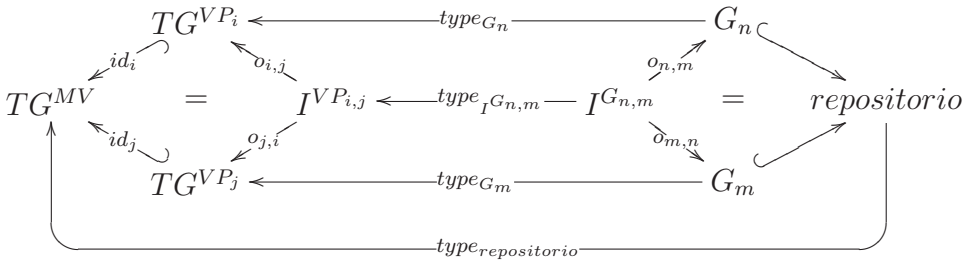


Figura 4.8: *Definición de un lenguaje visual de dominio específico multi-vista*

Sistemas de transformación de grafos triples para consistencia sintáctica

Para construir el repositorio a partir de las vistas del sistema, el presente enfoque hace uso de TGTSs que se derivan a partir de la especificación del LVDE multi-vista (grafos de tipos TG^{MV} y TG^{VP} en la definición anterior). Estos TGTSs proporcionan una implementación formal de las relaciones de inclusión de las vistas del sistema en el repositorio ($G \hookrightarrow repositorio$), y calculan implícitamente la intersección $I^{G_{n,m}}$ entre cada dos vistas. En concreto, cada vez que una vista del sistema se crea, elimina o cambia, se aplica un TGTS específico del tipo de vista que propaga los cambios al repositorio. Cuando el repositorio se actualiza debido a la creación o modificación de una vista, es posible que otras vistas pasen a un estado inconsistente si su intersección con la vista modificada no es vacía. Para recobrar la consistencia sintáctica se aplican otros TGTSs (también derivados a partir de la especificación del lenguaje) que propagan los cambios desde el repositorio al resto de vistas del sistema. De este modo para recuperar la consistencia sintáctica se sigue el patrón Modelo-Vista-Controlador: primero un TGTS propaga los cambios desde la vista creada/modificada/eliminada al repositorio, y a continuación un segundo conjunto de TGTSs propaga los cambios producidos en el repositorio al resto de vistas. La identificación de las partes comunes $I^{VP_{i,j}}$ entre cada dos puntos de vista ayuda a minimizar el número de reglas que toman parte en la propagación de cambios. Además, los TGTSs generados se pueden enriquecer con restricciones sintácticas adicionales específicas de dominio (también conocidas bajo el nombre de restricciones de la semántica estática), las cuales no son derivables de la especificación del lenguaje. Finalmente, para garantizar la consistencia de la semántica dinámica entre vistas del sistema se pueden utilizar transformaciones a otras vistas semánticas, tal y como se explicará en un apartado posterior.

Por cada punto de vista se genera automáticamente un TGTS específico que contiene las reglas de la figura 4.9 para cada nodo y arista concretos (con tipo no abstracto) del punto de vista. Estas reglas construyen o modifican el repositorio a partir de las vistas del sistema, y se aplican al grafo triple formado por la última vista modificada y el repositorio. La figura sólo muestra la estructura de las reglas para el tratamiento de nodos, ya que las de aristas son similares. Para identificar si dos elementos son el mismo, evalúan un atributo *id* que representa la clave (*keyword*) del tipo de elemento. De este modo, si un objeto aparece en dos vistas con el mismo *id* significa que son el mismo. Obviamente, el nombre del atributo clave en las reglas dependerá del nodo o arista concreto (por ejemplo, en las reglas generadas para una clase UML la comparación se realizaría por su nombre, que es clave). Las reglas para nodos y aristas sin clave sólo difieren en que no evalúan este *id*, y por tanto siempre consideran que un elemento sin clave es único en el sistema.

La primera regla de creación de la figura 4.9 añade un nodo al repositorio si dicho nodo es nuevo en la vista pero no está en el repositorio (NAC_1). La NAC_2 previene crear dos veces el mismo nodo en el repositorio cuando cambia el *id* del nodo que está en la vista.

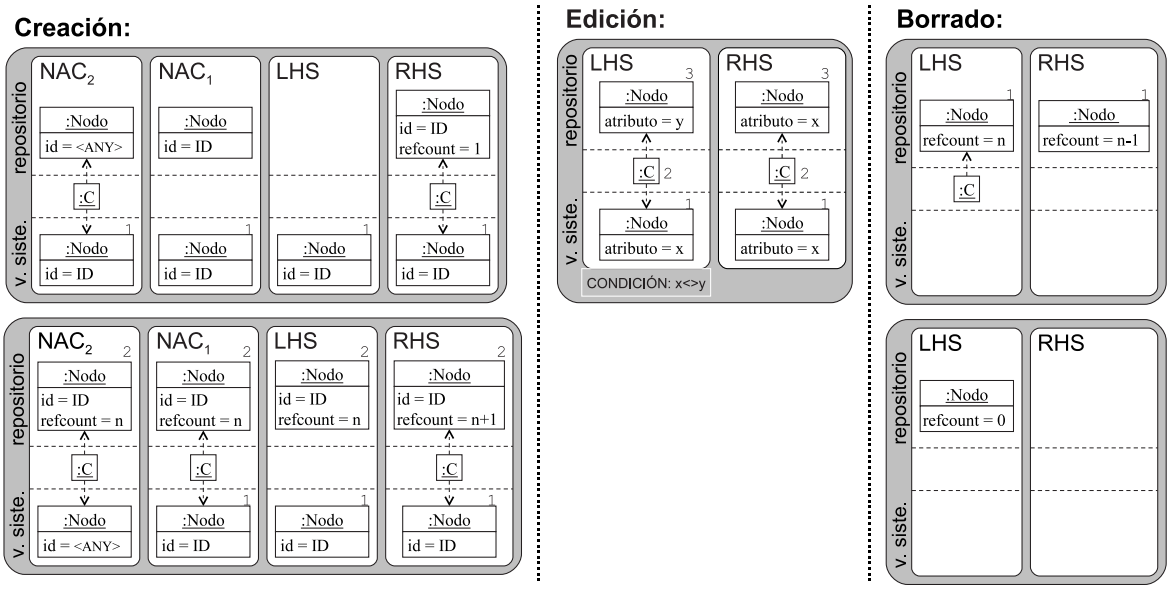


Figura 4.9: *Reglas triples para la construcción del repositorio*

Se considerará que el cambio de *id* está permitido sólo si su nuevo valor no está asignado a otro nodo del mismo tipo en el repositorio. La segunda regla de creación relaciona un nodo existente en una vista con el mismo nodo en el repositorio. Esto significa que el nodo se creó previamente en otra vista, desde donde se añadió al repositorio por medio de la aplicación de la primera regla. El atributo *refcount* que tienen todos los elementos del repositorio cuenta cuántas veces aparece un nodo en las distintas vistas del sistema. Cuando el nodo se añade al repositorio el atributo toma el valor 1; posteriormente, cada vez que el mismo nodo se añade a otra vista se incrementa su valor en una unidad. La NAC_2 de la regla se encarga de asegurar la confluencia del resultado en el siguiente escenario: el usuario cambia el *id* de un nodo que está relacionado con uno del repositorio, y a continuación crea un nuevo nodo en la vista con el *id* inicial del anterior. La NAC favorece la ejecución de la regla de edición antes que la de creación, asegurando que el resultado de aplicar el TGTS para una vista es siempre el mismo. Dicha regla de edición se encarga de copiar el valor de los atributos desde los nodos de las vistas a los correspondientes nodos en el repositorio.

Por último, la primera regla de borrado detecta cuándo se ha borrado un nodo de una vista (es decir, la función de correspondencia a la vista para el nodo del repositorio está indefinida), y resta una unidad al valor del atributo *refcount*. Cuando el atributo alcanza el valor 0, eso significa que el nodo no está siendo utilizado en ninguna vista, y por tanto la última regla de borrado lo elimina del repositorio.

Por cada punto de vista se genera un segundo TGTS que contiene una regla como la que muestra la figura 4.10 por cada nodo y arista concreto del punto de vista. La regla propaga cambios en el valor de los atributos desde el repositorio al resto de vistas del sistema. Estos

TGTSs se aplican tras un cambio en el repositorio, y se aplican a cada uno de los grafos triples formados por el repositorio y la vista a la que se realiza la propagación.

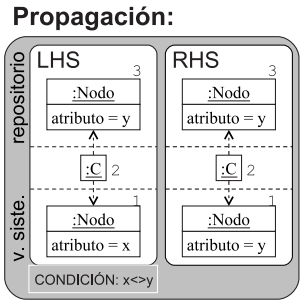


Figura 4.10: *Regla triple para la propagación de cambios*

La figura 4.11 muestra un ejemplo de propagación de cambios en un sistema formado por un diagrama de clases y una máquina de estados que tienen en común la clase `c1`. El repositorio del sistema se creó mediante la ejecución de los TGTSs de consistencia, así como los elementos de correspondencia entre sus elementos y los de cada una de las vistas. En el paso (i) del ejemplo el usuario modifica el nombre de la clase `c1`. Como consecuencia, en el paso (ii) se dispara la ejecución del TGTS asociado a los diagramas de clases para la construcción del repositorio, aplicándose sobre el grafo triple formado por el diagrama de clases modificado, el repositorio, y el grafo correspondencia entre ellos (zona coloreada en la imagen). De este modo la regla de edición copia el nuevo nombre de la clase desde la vista al repositorio. A continuación en el paso (iii) se aplican los TGTSs de propagación al resto de vistas, en este caso sobre el grafo triple formado por la máquina de estados, el repositorio y el grafo correspondencia entre ellos (zona coloreada en la imagen). Si hubiese otras vistas del sistema, se aplicaría el TGTS apropiado para el tipo de vista sobre el grafo triple formado por el repositorio y la vista en cada caso.

Análisis de la terminación y confluencia

Usar TGTSs para implementar los mecanismos de consistencia entre vistas permite verificar algunas propiedades interesantes de las transformaciones. Por ejemplo, en este caso resulta necesario demostrar que tales TGTSs tienen un comportamiento funcional, esto es, que el resultado que se obtiene de su aplicación es determinista (queremos que la ejecución del TGTS sobre un modelo determinado dé siempre el mismo resultado). Para ello se debe demostrar su terminación y confluencia [59, 61], tal como se hace a continuación para las reglas de construcción del repositorio (la terminación y confluencia para la regla de propagación de cambios es obvia).

En primer lugar, la ejecución del TGTS termina por lo siguiente:

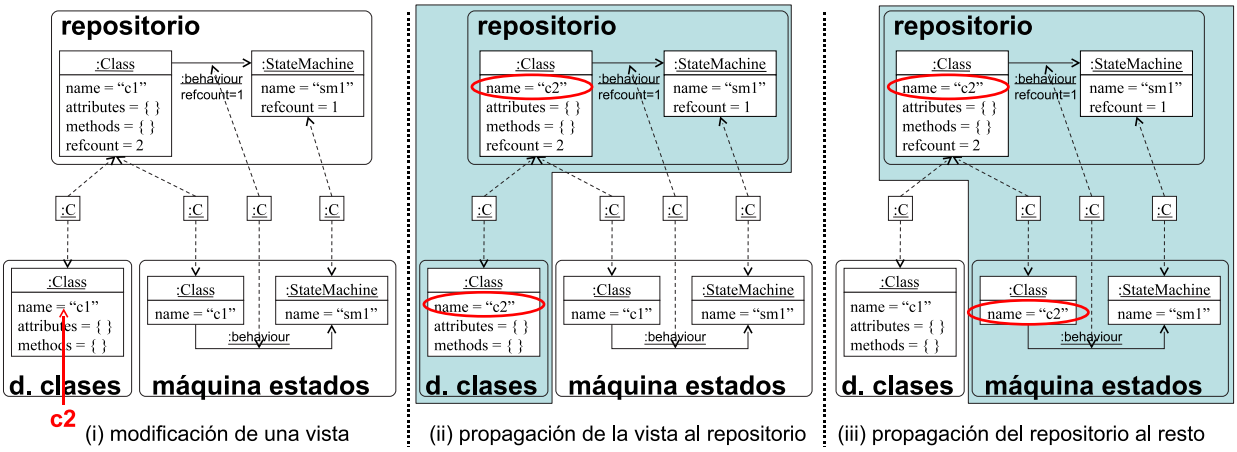


Figura 4.11: *Ejemplo de propagación de cambios*

1. Los elementos que crean las reglas de creación son también NACs. Esto asegura que las reglas se ejecutarán a lo sumo una vez por cada objeto de una vista.

2. De manera similar, aplicar una regla de edición hace que deje de cumplirse su condición de aplicación, y por tanto que sólo pueda ejecutarse una vez.

3. Las reglas de borrado sólo pueden aplicarse una vez ya que los elementos que eliminan son parte de la LHS de la regla. O en otras palabras, no se puede borrar dos veces el mismo objeto.

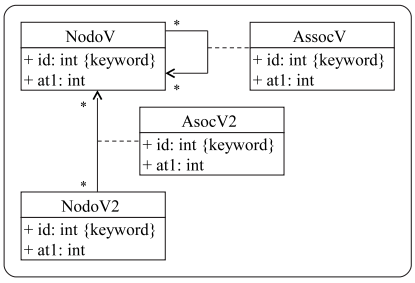
4. Un último motivo posible de no-terminación es que las reglas de borrado eliminen lo creado por las reglas de creación, y que éstas lo vuelvan a crear, y así recursivamente. Hay dos formas de demostrar que eso no puede ocurrir:

   - Una opción es dividir las reglas en dos capas que se ejecuten una a continuación de la otra [59]. Una capa contendría las reglas de creación y edición, y la otra las de borrado. Como la ejecución de cada capa por separado termina por las razones anteriormente expuestas, concluimos que el TGTS así organizado también termina.
   - Aun no dividiendo las reglas en capas, ejecutar cualquiera de las reglas de creación deshabilita la aplicación de las reglas de borrado sobre el elemento creado. Esto es así porque las primeras crean una función de correspondencia a la vista del sistema que es NAC de la primera regla de borrado (una NAC implícita derivada de la indefinición de dicha función en la LHS, lo cual es equivalente a la no-existencia de la función). La segunda regla de borrado tampoco se podría ejecutar porque necesitaría que el contador *refcount* hubiese disminuido hasta alcanzar el valor 0, lo cual sólo es posible mediante la ejecución de la primera

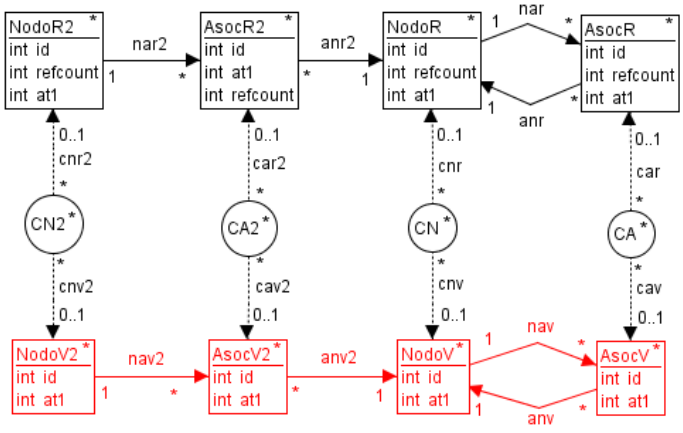
regla de borrado, que como ya se ha visto no sería aplicable. En consecuencia, no es posible entrar en un bucle donde los elementos que unas reglas crean las otras los borran, por lo que concluimos que el TGTS termina.

Entre las dos demostraciones es preferible la segunda por ser más general y no imponer un orden de ejecución de las reglas.

La confluencia del TGTS se puede verificar demostrando que todos sus pares críticos son confluentes. Hay que tener en cuenta que el TGTS generado es específico de cada meta-modelo ya que está formado por las reglas de creación, edición y borrado para cada uno de sus elementos. Aquí se va a demostrar la confluencia del TGTS que corresponde al meta-modelo de la figura 4.12(a). La extrapolación de los resultados a otros meta-modelos es directa ya que cualquier meta-modelo estará formado por una combinación de las estructuras básicas que éste contiene: nodos, relaciones entre nodos distintos, y relaciones de un nodo consigo mismo. El hecho de que nodos y relaciones puedan tener más o menos atributos no afecta a la demostración de confluencia.



(a) Meta-modelo del punto de vista



(b) Meta-modelo triple que relaciona el punto de vista y el repositorio

Figura 4.12: *Meta-modelos usados para la demostración de confluencia*

El cálculo de pares críticos se realizó en AGG tal como ilustraba la sección anterior. Así, primero hubo que especificar el grafo triple de tipos usado por las reglas del TGTS, el cual relaciona el punto de vista y el repositorio (sólo sus elementos relacionados con los del punto de vista). El grafo triple se muestra en la figura 4.12(b). Incluye los cambios pertinentes para poder representarlo en AGG: las funciones de correspondencia son relaciones y las asociaciones se modelan como nodos. A continuación se especificaron las 20 reglas del TGTS y se efectuó el análisis de pares críticos.

AGG detectó 47 pares críticos entre las 400 combinaciones posibles de pares de reglas, algunos de los cuales recoge la figura 4.13. Cada uno se estudió por separado para analizar su confluencia. Algunos pares críticos se descartaron porque el grafo necesario para que se produjera el conflicto no puede darse en una situación real, o bien por no poder llegar a producirse la secuencia de reglas que llevan al conflicto. Para el resto de casos se demostró su confluencia, concluyendo de ese modo la confluencia del TGTS.

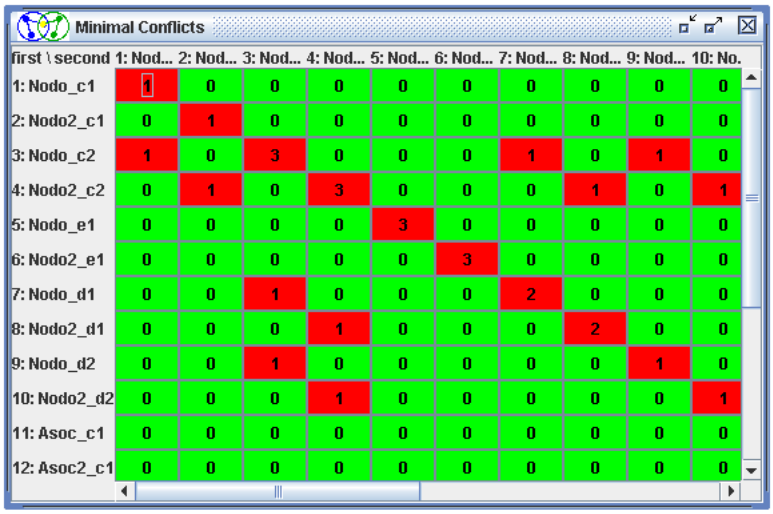


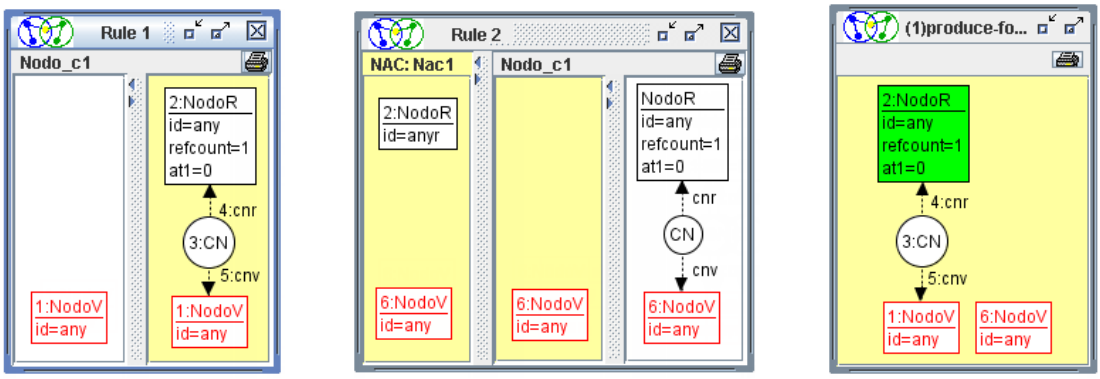
Figura 4.13: *Pares críticos detectados por AGG (mostrado parcialmente)*

A continuación se muestran los tipos de conflicto detectados y algún ejemplo representativo de cada uno de ellos.

1. Un primer tipo de conflicto detectado entre las reglas del TGTS fue cuando una crea elementos que son NAC de la otra, de tal modo que la segunda no puede ejecutarse. Esto originó los siguientes pares críticos:

   - todas las reglas de creación consigo mismas si intentan aplicarse sobre dos elementos distintos con igual identificador. Sin embargo esto no puede ocurrir nunca ya que el identificador de un elemento es único en un modelo, y por tanto estos pares críticos quedan descartados.

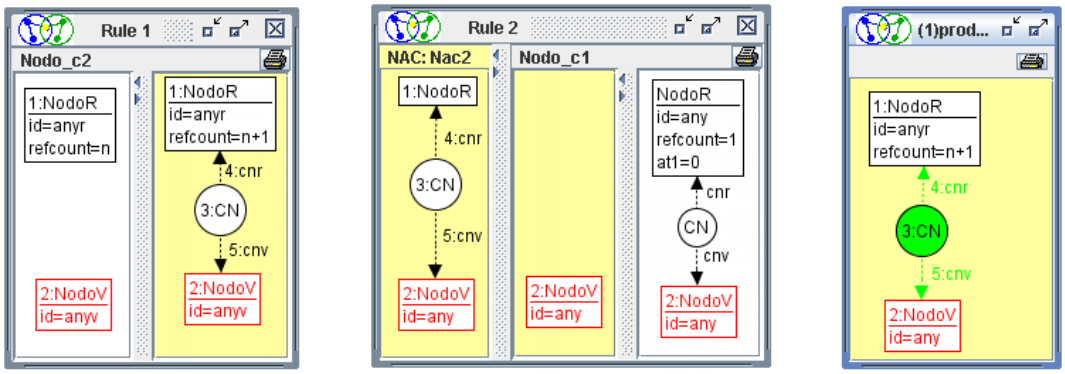
     La figura 4.14 muestra un conflicto de este tipo. Los elementos "2", "3", "4" y "5" del grafo (c) son resultado de la aplicación de la regla (a) sobre el grafo. En consecuencia, la regla (b) no puede aplicarse sobre el elemento "6" porque los elementos creados son NAC de la regla. Sin embargo esta situación no puede darse nunca porque implicaría que los nodos "1" y "6" tuviesen el mismo identificador, lo cual no puede ocurrir porque es un atributo de tipo clave.



(a) Primera regla (b) Segunda regla (c) Grafo conflicto

Figura 4.14: *Un par crítico del TGTS de consistencia*

- entre las reglas de creación del primer y segundo tipo si intentan aplicarse sobre el mismo elemento. La figura 4.15 muestra un par crítico de este tipo. Si partimos de un grafo de inicio que contiene un nodo de tipo *NodoV* y otro de tipo *NodoR* con el mismo identificador (esto es, $anyr == anyv$, lo que se comprueba con una condición sobre los atributos que no se muestra), al aplicar la regla (a) se obtiene el grafo que muestra (c). En este grafo no se puede ejecutar la regla (b) ya que no cumple la NAC.



(a) Primera regla (b) Segunda regla (c) Grafo conflicto

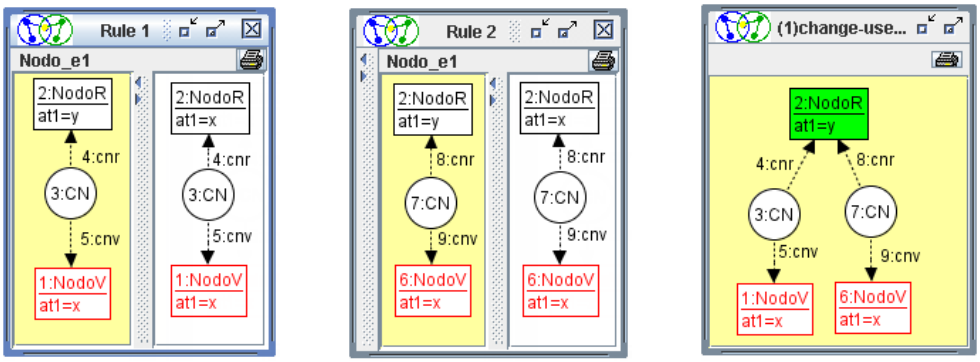
Figura 4.15: *Un par crítico del TGTS de consistencia*

Sin embargo, nótese que la regla (b) tampoco se habría podido aplicar antes de ejecutar (a), ya que tiene una NAC adicional a la mostrada en la regla que lo impide (véase definición de las reglas de consistencia en la figura 4.9, donde la NAC_1 de la primera regla de creación es igual a la LHS de la segunda regla de creación). Las dos reglas son disjuntas y no están en conflicto: para aplicar

una se necesita que haya un elemento con cierto *id* en el repositorio, mientras que para aplicar la otra se necesita que no lo haya. La regla (b) no habría podido aplicarse en (c) ni antes ni después de aplicar la regla (a), así que el comportamiento del par de reglas es determinista.

2. Otros conflictos se producían cuando una regla modificaba el valor de un atributo que la otra necesitaba para su aplicación. Esto originó pares críticos formados por reglas que o bien modificaban el atributo *refcount* (segunda regla de creación y primera de borrado) o bien el atributo *at1* (regla de edición). El estudio de los conflictos llevó a alguno de los dos resultados siguientes:

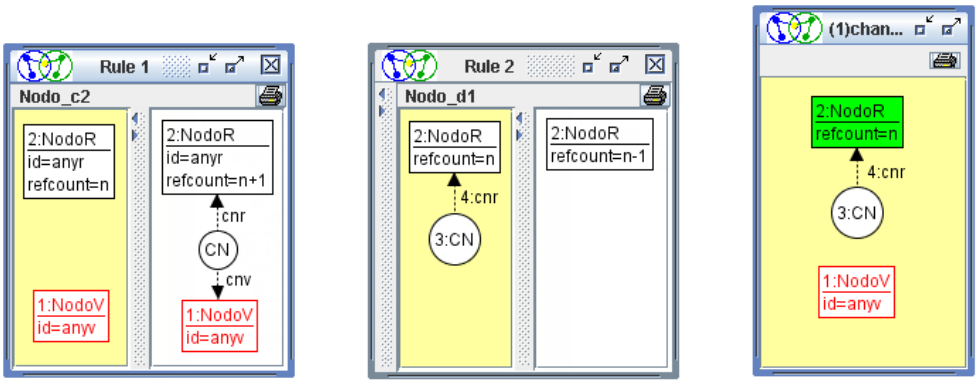
   - la mayoría de estos conflictos no podía llegar a producirse, y los pares críticos que originaban se descartaron. Por ejemplo, la figura 4.16 muestra un conflicto que surge al aplicar una regla de edición sobre dos nodos que están relacionados con el mismo nodo en el repositorio, situación que daría lugar no sólo a un resultado no confluente de la ejecución del TGTS sino a que ésta no terminase. Sin embargo no es posible que dos nodos de una misma vista estén relacionados con el mismo nodo en el repositorio porque para ello deberían tener el mismo identificador, lo cual es imposible.



(a) Primera regla (b) Segunda regla (c) Grafo conflicto

Figura 4.16: *Un par crítico del TGTS de consistencia*

   - en el resto de casos el resultado de aplicar las reglas era confluente. Por ejemplo, la figura 4.17 muestra uno de esos casos que corresponde a la ejecución de una regla de creación que incrementa el atributo *refcount* una unidad, y posteriormente la ejecución de una regla de borrado que lo disminuye. Aplicar una de las reglas no impide aplicar la otra ya que ninguna borra elementos necesarios para la otra. Además, el resultado es confluente ya que da igual sumar uno al atributo *refcount* y luego restárselo, o hacerlo al contrario.

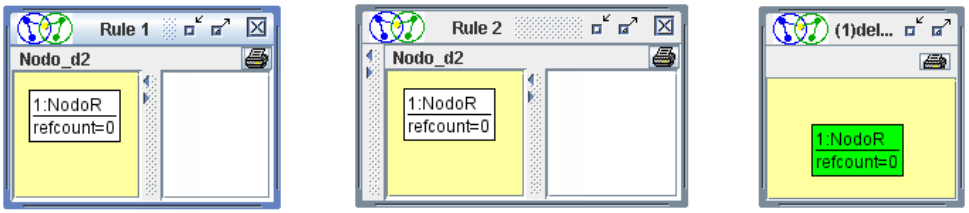


(a) Primera regla (b) Segunda regla (c) Grafo conflicto

Figura 4.17: *Un par crítico del TGTS de consistencia*

3. El último motivo de conflicto fue que una regla borrase elementos que otra necesitaba para su aplicación. Esto originó los siguientes pares críticos:

   - todas las reglas de borrado consigo mismas si intentan aplicarse sobre el mismo elemento, siendo en todos los casos confluentes. Por ejemplo, la figura 4.18 muestra uno de estos casos. Aplicar la regla (a) sobre el grafo (c) elimina el nodo que éste contiene, de tal modo que no es posible aplicar la regla (b) porque ésta necesita dicho nodo para poder ejecutarse. En cualquier caso el resultado siempre confluye ya que la regla se puede aplicar una sola vez con idéntico resultado.

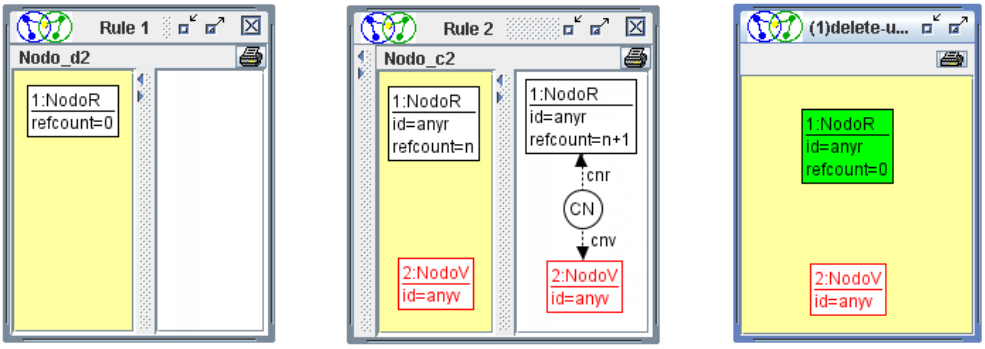


(a) Primera regla (b) Segunda regla (c) Grafo conflicto

Figura 4.18: *Un par crítico del TGTS de consistencia*

   - el segundo tipo de reglas de borrado con el segundo tipo de reglas de creación, tal como muestra el conflicto de la figura 4.19. Aplicar la regla (a) sobre el grafo (c) borra el nodo del repositorio, de modo que si existe un nodo en la vista con el mismo identificador no se puede aplicar sobre él la regla (b). En su lugar se ejecutaría el primer tipo de regla de creación añadiendo un nodo al repositorio con el mismo identificador, atributo *refcount* a 1, y relacionado con el nodo de la vista. Si por el contrario se aplica primero la regla (b) sobre el grafo (c) el

resultado sería el mismo, ya que la regla relacionaría el nodo de la vista y el del repositorio e incrementaría el atributo *refcount* de esta última, pasando a valer 1. En ese caso ya no sería posible aplicar la regla (a). Por tanto da igual aplicar primero la regla (a) y después la (b), o viceversa, ya que el resultado es confluente.



(a) Primera regla (b) Segunda regla (c) Grafo conflicto

Figura 4.19: *Un par crítico del TGTS de consistencia*

Por tanto queda demostrada la terminación del TGTS y la confluencia de sus pares críticos, y de este modo su comportamiento funcional.

Patrones de comportamiento configurable

Los TGTSs generados a partir de la descripción del LVDE multi-vista fijan la política de consistencia seguida por el lenguaje y definen su comportamiento. Sin embargo, distintos lenguajes pueden requerir distintas políticas. Por esta razón resulta interesante poder elegir el patrón de comportamiento más adecuado en cada caso, y generar distintos conjuntos de reglas dependiendo del comportamiento que se quiera obtener.

Por ejemplo, las reglas de borrado de la figura 4.9 realizan un borrado conservativo donde los elementos del repositorio sólo se eliminan si no aparecen en ninguna vista del sistema. Sin embargo, un patrón de comportamiento alternativo es el borrado en cascada, donde eliminar un elemento de una vista provoca su eliminación de todas las vistas del sistema donde aparece, así como del repositorio. Tal comportamiento se puede obtener reemplazando las reglas de borrado anteriores por las de la figura 4.20. Las tres reglas de la izquierda pertenecen al TGTS que construye el repositorio. Las dos primeras se generan para cada nodo y arista de la vista, y son disjuntas. La primera detecta si un elemento se ha eliminado de una vista (es decir, la función de correspondencia para el elemento del repositorio no está definida en la vista), pero el usuario lo ha vuelto a crear (razón por la cual la vista contiene un elemento con ese *id*). En ese caso no hay que realizar el

borrado en cascada porque el elemento sigue existiendo, así que la regla borra el nodo de correspondencia con la función indefinida y deja que las reglas de creación del TGTS lo regeneren. En cambio, la segunda regla se aplica si el elemento se ha eliminado totalmente de la vista (NAC), y entonces lo elimina del repositorio. La propiedad de *enlaces colgantes* no permite aplicar esa regla sobre nodos que tienen aristas de entrada o salida. Por ello un conjunto de reglas como la tercera de la figura se encarga de borrarlas. La regla se genera para cada posible arista de entrada o salida al nodo.

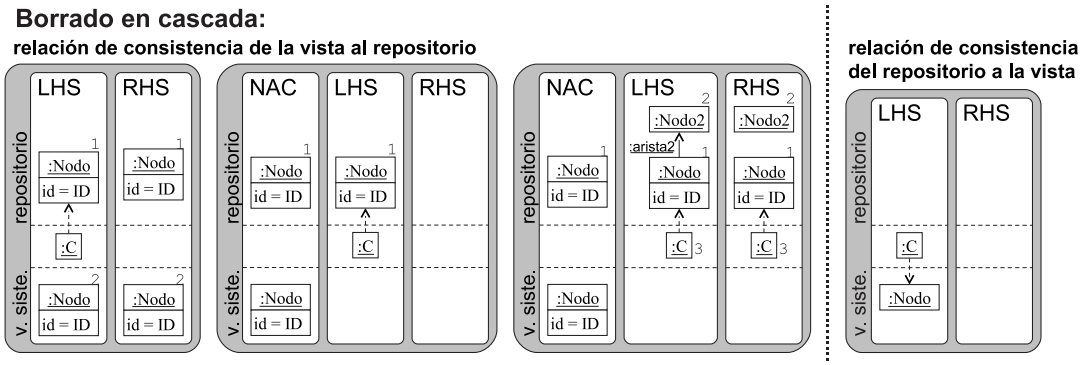


Figura 4.20: *Reglas triples para borrado en cascada*

Nótese que las reglas expuestas se aplican al grafo triple formado por la vista donde se eliminó el elemento y el repositorio; aunque el elemento se use en otras vistas, las funciones de correspondencia de esas otras vistas no se incluyen en este grafo triple, sino que se tratan en los TGTSs para la propagación de cambios. En estos TGTSs se incluye la cuarta regla de la misma figura, generada para cada nodo y arista de la vista a la que los cambios se propagan. La regla borra de la vista un elemento si no está en el repositorio. Como puede verse, el atributo *refcount* es innecesario en este patrón de comportamiento.

Obviamente, al modificar los TGTSs de consistencia con las nuevas reglas de borrado hay que volver a demostrar que el resultado que se obtiene de su aplicación termina y es confluente. Demostrar la terminación es trivial porque las reglas de borrado sólo eliminan elementos, proceso que acaba siempre. Tampoco es posible entrar en un bucle de ejecución infinito con las reglas de creación por las razones expuestas para las reglas básicas de consistencia. En cuanto a la confluencia, se utilizó AGG pero considerando sólo los pares de reglas que incluían alguna de borrado en cascada. De las 432 posibles combinaciones de reglas para el meta-modelo usado, sólo 28 eran pares críticos. Éstos correspondían a alguno de los siguientes casos que, o bien no podían ocurrir, o eran confluentes, demostrando de ese modo la confluencia del TGTS:

- todas las reglas de borrado consigo mismas son pares críticos ya que, al aplicarlas, borran elementos necesarios para su nueva aplicación. Esto no afecta a la confluencia del TGTS ya que el resultado es siempre la eliminación de cierto elemento.

- AGG detecta como pares críticos las reglas de borrado del primer tipo y las del segundo. Sin embargo eso no es cierto ya que ambas reglas son disjuntas: la primera necesita que haya un elemento con cierto *id* en la vista, mientras que la segunda necesita que no lo haya. Por tanto no hay conflicto, pues sólo una de las reglas es aplicable en cada momento. Por la misma razón tampoco son pares críticos (a pesar de ser detectados por AGG) la tercera regla de borrado en cascada y la segunda de creación, ni la primera y tercera de borrado.
- El último conflicto surge cuando se borran el origen y el destino de una arista, ya que hay dos reglas del tercer tipo que permiten borrarla (una generada para el nodo origen y otra para el destino). Aplicar una de las reglas borra la arista e impide que se aplique la otra. En cualquier caso la acción de ambas reglas es la misma, y por tanto el resultado es confluente.

Por otro lado, los TGTSs mostrados hasta ahora pueden ejecutarse tanto de manera asíncrona (tras crear o modificar una vista y validar los cambios) como síncrona (en respuesta a una acción del usuario, como crear un elemento). Sin embargo existen patrones de comportamiento que son intrínsecamente síncronos, como es la creación "inteligente". En este patrón la creación de un nuevo elemento en una vista dispara la ejecución de una regla que copia los atributos desde el mismo elemento en el repositorio (si existe) a la vista. Para ello se genera una regla como la que muestra la figura 4.21 por cada tipo de nodo y arista en el meta-modelo del LVDE completo. La regla asociada al tipo del nuevo nodo o arista se ejecuta inmediatamente después de crear el elemento y especificar su *id*. Como la ejecución se realiza una sola vez, la demostración de terminación y confluencia es trivial.

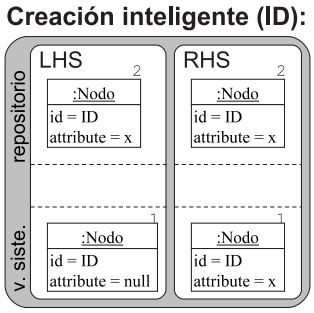


Figura 4.21: *Regla triple para creación inteligente*

Con reglas síncronas también se puede hacer que cualquier cambio de *id* para un elemento sea válido, eliminando así la restricción de que el nuevo valor no pueda estar en el repositorio. Esta restricción se definió para evitar problemas de confluencia que derivaran en la creación de dos elementos con el mismo *id* en el repositorio, pero impone una limitación en la forma de construir los modelos. Las reglas de la figura 4.22 resuelven este

problema. Se generan para cada nodo y arista, de tal modo que las que corresponden a un tipo determinado se ejecutan inmediatamente después de editar un elemento de ese tipo (esto es, son reglas síncronas). La primera regla corresponde a la edición de un nodo, y detecta si su *id* ha cambiado a un valor existente en el repositorio, único caso no permitido por el TGTS. De ser así, simula el borrado del nodo con el *id* antiguo eliminando la relación de correspondencia entre los nodos de la vista y el repositorio, y disminuyendo el contador de este último una unidad. Posteriormente, los TGTSs asíncronos de consistencia considerarán que el nodo es nuevo, y las reglas de creación lo relacionarán con el elemento adecuado del repositorio. Además, como borrar un nodo implica borrar sus aristas, también se generan reglas como la segunda de la figura que simulan el borrado de las aristas de entrada y salida del nodo modificado. De hecho, la primera regla no debe poder aplicarse hasta haber borrado todas las aristas del nodo, de lo cual se encargan NACs adicionales (una por cada tipo de arista entrante y saliente del tipo del nodo) como la que muestra la figura.

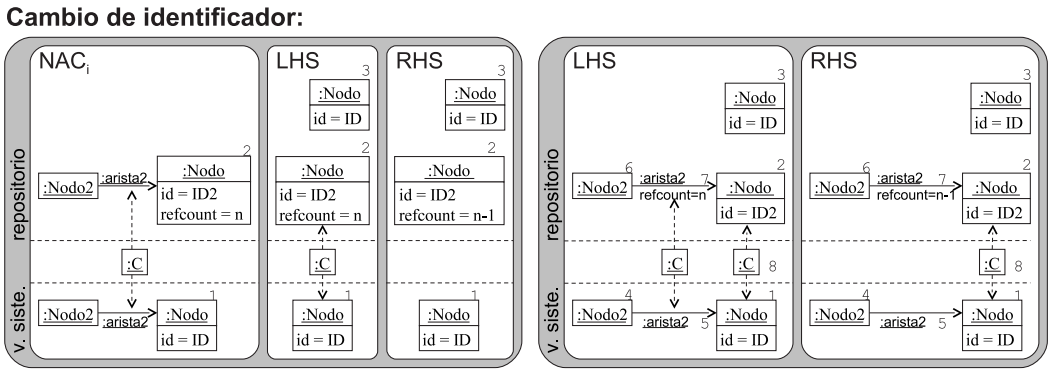


Figura 4.22: *Reglas triples para cambio de identificador*

La terminación de las reglas para el cambio de identificador se deduce del hecho de que todas son reglas de borrado, así que su ejecución es finita. En cuanto a su confluencia, AGG identificó como pares críticos a todas las reglas consigo mismas si se intentan aplicar sobre los mismos elementos de correspondencia, ya que sólo pueden borrarse una vez. Por la misma razón, también son pares críticos las reglas generadas para aquellas aristas que tienen como origen y destino el mismo nodo, ya que en ese caso el borrado puede realizarlo la regla donde la arista desempeña el rol de origen de la arista, o donde desempeña el rol de destino. En cualquier caso, todos los pares críticos son confluentes.

4.1.3. Vistas semánticas

Las vistas semánticas recogen toda (o parte de) la semántica del sistema expresada en un formalismo que dispone de herramientas de análisis, como por ejemplo redes de Petri [145] o álgebra de procesos [15]. Se utilizan para verificar la consistencia de la semántica dinámica del sistema, para analizar sus propiedades o para simular su comportamiento. Con este objetivo la parte del sistema a verificar se transforma a un dominio semántico donde se realiza el análisis de interés, y posteriormente los resultados obtenidos se anotan de nuevo al modelo original de tal modo que puedan darse en términos de la notación usada para la especificación del sistema. De este modo, la definición de vistas semánticas permite integrar técnicas de análisis y verificación desde la misma especificación del lenguaje multi-vista. Además, como los mecanismos de anotación están integrados en el proceso de análisis, el futuro usuario del lenguaje no necesita conocer las formalidades matemáticas o lógicas del dominio semántico utilizado, ya que los resultados de los análisis se devuelven en la notación original.

Para ello, el presente enfoque permite enriquecer la definición de un LVDE multi-vista con la definición de vistas semánticas para los puntos de vista del lenguaje o su repositorio. La sintaxis de la vista semántica se expresa mediante un meta-modelo independiente, que no tiene por qué ser una proyección del meta-modelo del LVDE. La relación entre la vista semántica y el punto de vista o repositorio se especifica mediante un TGTS que establece cómo crear la vista semántica que captura el comportamiento del punto de vista (o de parte de él), así como los elementos de correspondencia que surgen entre ambos. Una misma vista semántica puede servir para verificar distintas propiedades del modelo a partir del que se generó. Por ejemplo, una red de Petri puede servir tanto para estudiar la alcanzabilidad de un estado como para analizar la terminación de un sistema. Por esa razón, para cada tipo de propiedad a verificar en el modelo origen se especifica una llamada a un método de análisis específico que se ejecutará en el dominio semántico resultante de la transformación, y cuyo resultado se devolverá al modelo original gracias a los elementos de correspondencia creados durante dicho proceso de transformación. La definición de tales métodos de análisis puede requerir también la definición de un proceso previo de adquisición de parámetros necesarios para el método de análisis, así como de uno posterior para procesar los resultados obtenidos.

Respecto al mecanismo de anotación de resultados desde el dominio semántico al modelo origen, aplicar la inversa del TGTS resulta insuficiente en un caso general, ya que restringe el enfoque a la anotación de un solo elemento del modelo origen por cada elemento de la vista semántica. Sin embargo, a veces un resultado en el dominio semántico se tiene que reflejar en varios elementos del modelo origen y no sólo en el que está relacionado a través del elemento de correspondencia. Otras veces el resultado obtenido tras el análisis no es un elemento del modelo semántico, sino que es un valor booleano, un número o una matriz,

por ejemplo. En esos casos la anotación no se puede conseguir simplemente siguiendo los enlaces creados en el proceso de transformación, ya que quizás el resultado se recoge mejor en un informe o cuadro de diálogo. Por último, un análisis semántico puede devolver varias soluciones que deben mostrarse en el modelo origen una cada vez, y no simultáneamente.

Por estas razones el presente enfoque utiliza TGTSs unidireccionales para realizar la transformación, y la anotación de resultados se expresa mediante patrones triples de anotación que especifican gráficamente cómo el resultado de un análisis se representa en el modelo original, superando de ese modo la limitación de anotaciones 1-a-1. En aquellos casos en que el resultado no es representable en el modelo origen, sino que debe darse como un informe o cuadro de diálogo (por ejemplo cuando el resultado es un valor booleano o una matriz de valores), se puede utilizar la fase de post-procesamiento para codificar la anotación adecuada en cada caso.

Patrones triples de anotación

Un patrón triple de anotación consta de un *grafo triple positivo*, y puede contener un conjunto de condiciones de aplicación formadas por un grafo triple premisa y un conjunto de grafos triples consecuencia (al estilo de las condiciones de aplicación utilizadas en transformación de grafos [61, 93]). La aplicación de un patrón triple a un grafo triple obtiene los subgrafos de este último que son isomorfos al grafo positivo del patrón, y que además cumplen las condiciones de aplicación. Opcionalmente, el patrón puede recibir como parámetro un conjunto de elementos del grafo triple al que se aplica para inicializar la búsqueda, y el resultado puede filtrarse para obtener un subgrafo de cada resultado obtenido. En el caso presente, los argumentos del patrón son los elementos del dominio semántico que se quieren anotar, y la salida son los elementos del modelo original resultantes del proceso de anotación.

Por ejemplo, la figura 4.23 muestra un ejemplo de patrón triple de anotación. Su grafo triple positivo relaciona una máquina de estados (el modelo origen) y una red de Petri coloreada (el dominio semántico). El patrón busca dos estados en la máquina de estados y una transición entre ellos que esté relacionada con una transición de la red de Petri. El argumento del patrón es el elemento con etiqueta “1” (la transición de la red de Petri), y la salida son los elementos etiquetados “2”, “3” y “4” (la transición y los estados de la máquina de estados). Si un método de análisis devuelve una transición de red de Petri como resultado, ésta pasa al patrón triple de anotación como parámetro, y entonces la correspondiente transición de la máquina de estados junto con sus estados origen y destino son el resultado equivalente en términos del modelo origen.

La figura 4.24 muestra la anotación del resultado de un análisis en el dominio semántico de las redes de Petri coloreadas al modelo original. Para ello se utiliza el patrón de la figura 4.23. En el paso (i) comienza la búsqueda del patrón en el grafo triple G a partir del

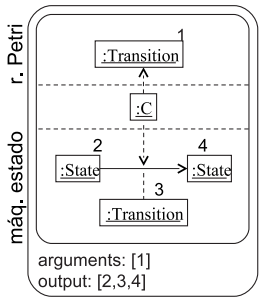


Figura 4.23: *Ejemplo de patrón triple de anotación*

elemento que se quiere anotar, el cual es parámetro del patrón. En la figura, el elemento a anotar es la transición de la red de Petri que se muestra coloreada. A continuación, en el paso (ii), el subgrafo de G que contiene el elemento inicial se amplía hasta contemplar el grafo triple positivo. En el ejemplo, G sólo contiene un subgrafo isomorfo al grafo positivo del patrón, aunque en un caso general puede haber varios o no encontrarse ninguno. Finalmente, en el paso (iii) se filtran los subgrafos encontrados para devolver sólo los elementos que actúan como salida del patrón. De este modo el proceso de anotación únicamente devuelve la parte de la máquina de estados identificada por el patrón.

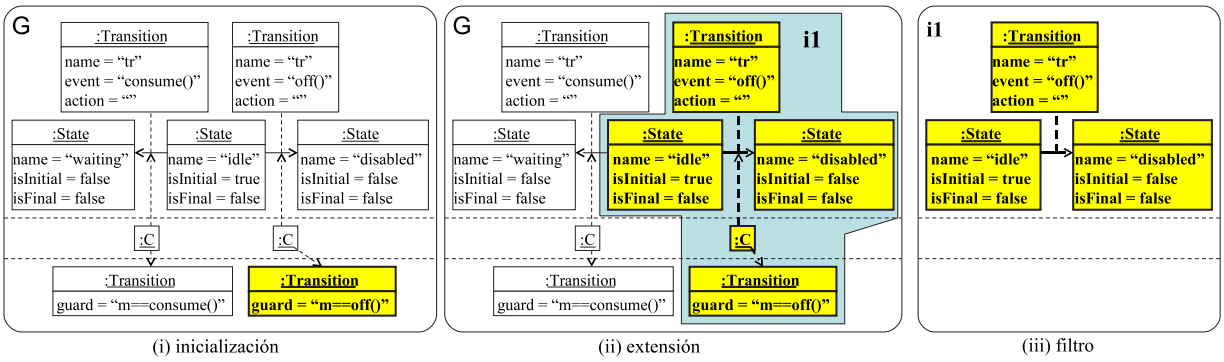


Figura 4.24: *Ejemplo de anotación de un resultado mediante el patrón triple de la figura 4.23*

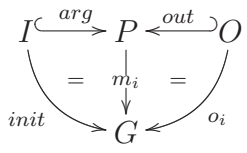
Cabe señalar que si un método de análisis específico devuelve no uno, sino un conjunto de elementos como resultado (por ejemplo, las transiciones de una máquina de estados que nunca se ejecutan), el patrón se aplicaría tantas veces como fuese necesario, una vez por cada uno de los elementos a anotar. Por otro lado, si el método de análisis devuelve distintos resultados que tienen que ser mostrados separadamente (por ejemplo, todas las secuencias de transiciones que llevan a un estado determinado de una máquina de estados), la anotación de cada resultado se realiza para cada posible solución por separado, utilizando los patrones de anotación para los elementos de una solución cada vez.

Formalmente, un patrón triple de anotación se define como $p = (P, I \overset{arg}{\hookrightarrow} P, O \overset{out}{\hookrightarrow}$

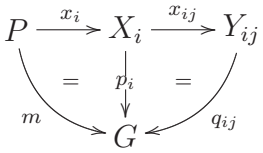
$P, \bigwedge_{i \in I}(x_i \Rightarrow \vee_{j \in J_i} x_{i,j}))$, donde:

- P es el grafo triple positivo,
- I es el subgrafo triple de P que contiene los parámetros de entrada,
- O es el subgrafo triple de P que contiene la salida, y
- $x_i : P \to X_i$ y $x_{i,j} : X_i \to Y_{i,j}$ son aplicaciones inyectivas (donde X_i e $Y_{i,j}$ se denominan grafos triples premisa y consecuencia respectivamente).

La figura 4.25 muestra dos diagramas que explican el proceso de aplicación de un patrón triple a un grafo triple. Aunque los diagramas se muestran separados por motivos de legibilidad, forman parte del mismo proceso. El diagrama de la izquierda muestra un patrón triple P sin condiciones de aplicación, con entrada I y salida O (ambos subgrafos de P). Al aplicar P sobre un grafo G primero se obtienen todas las funciones $m_i \colon P \to G$ que conmutan con la función de inicialización $init \colon I \to G$. Entonces, el resultado de aplicar el patrón son las funciones $o_i \colon O \to G$ tales que el triángulo derecho de la figura conmuta con la función m_i correspondiente. La figura 4.25(b) muestra cómo se evalúan las condiciones de aplicación. En este diagrama se han excluido los grafos triples I y O para mayor claridad, aunque estarían presentes. Dada una función $m \colon P \to G$, si existe una función $p_i : X_i \to G$ entonces también debe existir una función $q_{i,j} : Y_{i,j} \to G$ que haga conmutar los dos triángulos de la figura, para que m se considere un resultado válido de la aplicación del patrón triple sobre G.



(a) Grafo triple positivo



(b) Condiciones de aplicación

Figura 4.25: *Formalización de la aplicación de un patrón triple a un grafo triple*

Técnicamente, las funciones m, p_i y $q_{i,j}$ son funciones-clan (del inglés *clan-morphism* [61]), lo que significa que pueden relacionar elementos de P, X_i y $Y_{i,j}$ con objetos de cualquier tipo que pertenezca a su jerarquía de herencia (es decir, con elementos que tengan su mismo tipo o cualquiera de sus subtipos). Además, existe la restricción de que el tipo de los elementos en $Y_{i,j}$ debe ser igual o más concreto que el tipo de los elementos en X_i, y a su vez éstos deben ser igual o más concretos que los de P.

Como puede verse, el concepto de patrón triple presentado es similar a la noción de restricción de grafo (*graph constraint* [61, 93]) utilizada en el dominio de la transformación de grafos, salvo por los conceptos de inicialización (I) y filtrado (O).

4.1.4. Vistas derivadas y dirigidas a audiencia

Durante el proceso de especificación de un sistema, además de proporcionar información sobre el mismo en forma de vistas del sistema también existe la necesidad de extraer información de él como resultado de una consulta. La consulta se puede realizar bien sobre una vista del sistema, o bien sobre el repositorio si la información a extraer está dispersa en varias vistas. El resultado es un subgrafo de la vista o del repositorio que se denomina vista derivada si la consulta la realiza el usuario del lenguaje, o vista dirigida a audiencia si por el contrario la consulta viene predefinida con la especificación del lenguaje. Estas últimas son útiles para mostrar en un solo modelo la información relevante para un cierto tipo de usuario, quizás utilizando una sintaxis concreta distinta adaptada al mismo.

En el presente enfoque, las consultas se especifican utilizando patrones de consulta visuales y declarativos que se evalúan sobre un modelo base G y dan como resultado una vista derivada (o dirigida a audiencia) V^Q. La vista se construye mediante un TGTS (generado automáticamente a partir del patrón) que se aplica al grafo triple formado por el modelo base y la vista derivada a construir. El TGTS también proporciona mecanismos de sincronización para reflejar cambios posteriores realizados en el modelo base sobre la vista resultante de la consulta.

Patrones visuales de consulta

Formalmente, un patrón visual de consulta $Q = (TG^Q, \{(P_i^Q, P_i)\}_{i\in I}, \{(N_j^Q, N_j)\}_{j\in J})$ se compone de:

- un meta-modelo TG^Q con la sintaxis de la vista derivada a obtener con la consulta. Este meta-modelo debe ser un subconjunto del meta-modelo TG del modelo base.
- un conjunto de restricciones positivas P_i sobre algún elemento P_i^Q del meta-modelo TG^Q. Cada restricción positiva especifica un patrón que debe estar presente en el modelo base para que el elemento al que afecta la restricción pase a formar parte del resultado de la consulta. Si varias restricciones afectan al mismo elemento se realiza su disyunción (al menos una se tiene que cumplir), mientras que si afectan a elementos distintos se toma su conjunción (todas deben cumplirse).
- un conjunto de restricciones negativas N_j sobre algún elemento N_j^Q del meta-modelo TG^Q. Estas restricciones son patrones que no deben cumplir los elementos del modelo base para formar parte del resultado de la consulta. Ninguna de ellas se debe cumplir, afecten a elementos del mismo o de distinto tipo.

Por ejemplo, la figura 4.26 muestra un patrón visual de consulta para obtener un vista derivada con las clases y relaciones de herencia entre ellas, lo que se modela con el grafo

TG^Q del patrón. Nótese que no se está interesado en obtener los métodos de las clases, por lo que TG^Q no incluye ese atributo en la clase *Class*. Además, la consulta sólo obtiene aquellas clases que no tienen asociada una máquina de estados (restricción negativa N_1) pero que tienen una clase padre (restricción positiva P_1).

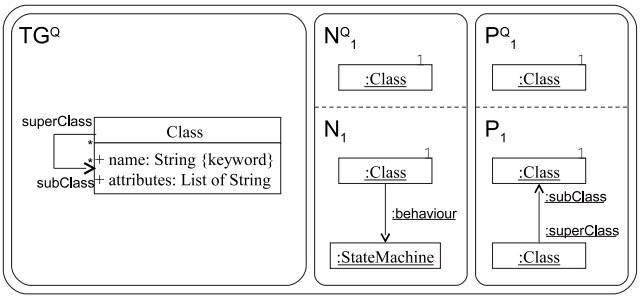


Figura 4.26: *Ejemplo de patrón visual de consulta*

La figura 4.27 muestra el proceso de aplicación de un patrón visual de consulta a un modelo base G. El tipo de las restricciones, que no se muestra por claridad, es TG para los grafos P_i y N_j, y TG^Q para los grafos P_i^Q y N_j^Q. Para aplicar el patrón al grafo, primero se calcula el objeto *pullback* V^G de $id_{TG^Q}\colon TG^Q \to TG$ y $type_G\colon G \to TG$ (cuadrado (1) en la figura). V^G es el subconjunto de G restringido por el meta-modelo TG^Q. A continuación V^G se restringe para contener sólo aquellos elementos que cumplen las restricciones positivas y negativas especificadas en el patrón de consulta, dando como resultado la vista derivada V^Q. De este modo, para todos los morfismos $p_{il}^Q\colon P_i^Q \to V^Q$ debe existir un morfismo p_{il} de la restricción P_i a G tal que el cuadrado (2) de la figura conmute. Además, para ningún morfismo $n_{jk}^Q\colon N_j^Q \to V^Q$ debe existir un morfismo n_{jk} de la restricción N_j a G tal que el cuadrado $id_{V^G} \circ id_{V^Q} \circ n_{jk}^Q = n_{jk} \circ id_{N_j^Q}$ conmute. En otras palabras, para incluir un elemento x con tipo $type_{V^Q}(x)$ en la vista derivada V^Q, éste debe cumplir alguna de las restricciones positivas y ninguna de las negativas definidas para el tipo. Nótese que si el patrón no define restricciones sobre el tipo, el elemento pertenecerá a la vista derivada (si éste pertenece a TG^Q).

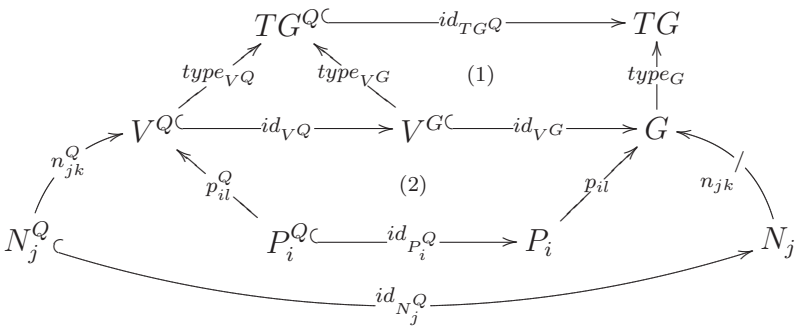


Figura 4.27: *Patrón visual de consulta aplicado a un grafo G, obteniendo la vista derivada V^Q*

La expresividad de los patrones visuales de consulta así definidos es igual al fragmento $\exists\neg\exists$ de lógica de primer orden, limitada por la anidación permitida en P_i y N_i que viene dada por ser traducción a condiciones de aplicación aplicables a un elemento. Esto es similar a la expresividad de las restricciones sobre grafos (*graph constraints*) [156].

Sistemas de transformación de grafos triples para patrones visuales de consulta

A partir de un patrón de consulta se genera un TGTS que construye la vista derivada a partir del modelo base, y posteriormente propaga cualquier cambio realizado sobre el modelo base a la vista derivada. Dicho TGTS implementa el proceso de aplicación de un patrón a un grafo que mostraba de manera conceptual la figura 4.27. El TGTS se ejecuta sobre un grafo triple que tiene G como grafo origen, y tiene la vista derivada V^Q (inicialmente vacía, y tipada sobre el componente TG^Q del patrón) como grafo destino. Sus reglas se parecen a las usadas para construir el repositorio a partir de las vistas del sistema, pero incluyen además condiciones de aplicación adicionales derivadas de las restricciones P_i^Q y N_j^Q del patrón.

En concreto, el conjunto de reglas triples generadas a partir de un patrón visual Q que se aplica sobre un modelo base con meta-modelo TG se expresa como $TGTS(Q, TG) = \{C^N, C^E, E^N, E^E, D^N, D^E\}$. Las letras C, E y D hacen referencia a conjuntos de reglas de creación, edición y borrado respectivamente, y los superíndices N y E hacen referencia a nodos y aristas. Para crear la vista derivada a partir del modelo base sólo se utilizan los conjuntos de reglas C^i y E^i, pero se necesita todo el TGTS para sincronizar la vista derivada creada respecto a cambios en el modelo base. El conjunto de reglas C^N copia los nodos de los tipos especificados en TG^Q desde el modelo base a la vista derivada, mientras que las del conjunto C^E copian las aristas. Las restricciones positivas y negativas del patrón Q se usan para restringir la aplicabilidad de estas reglas. Los conjuntos E^N y E^E contienen las reglas que copian el valor de los atributos desde los nodos y aristas del modelo base a los de la vista derivada. Finalmente, las reglas de D^N y D^E borran los nodos y aristas de una vista derivada previamente creada cuando éstos se eliminan del modelo base o dejan de cumplir las restricciones impuestas por el patrón de consulta.

La figura 4.28 muestra las reglas básicas (sin considerar las restricciones del patrón) que se derivan a partir de un patrón de consulta. En total se genera una regla de cada tipo para cada nodo y arista especificados en el grafo TG^Q de la consulta. A modo de ejemplo, la figura muestra las reglas generadas para el nodo de tipo *Class*. La primera regla pertenece al conjunto C^N y copia los elementos del modelo base a la vista derivada si éstos aún no se encuentran en ella. La segunda regla pertenece al conjunto E^N y propaga el valor de los atributos desde los elementos del modelo base a la vista derivada. La última regla pertenece a D^N y elimina un elemento de la vista derivada si éste no aparece en el modelo base (esto es, si la función de correspondencia al modelo base está indefinida).

Como se dijo anteriormente, sólo las reglas de los conjuntos C^i y E^i se usan para crear la vista derivada. Si posteriormente el modelo base cambia, es posible que el resultado de la consulta se vea afectado bien porque haya que añadir nuevos elementos que cumplen el patrón, o porque haya que eliminar otros que han dejado de cumplirlo. En el primer caso las reglas de C^i y E^i se encargan de añadir los nuevos elementos, mientras que para el segundo caso se necesitan las reglas de D^i.

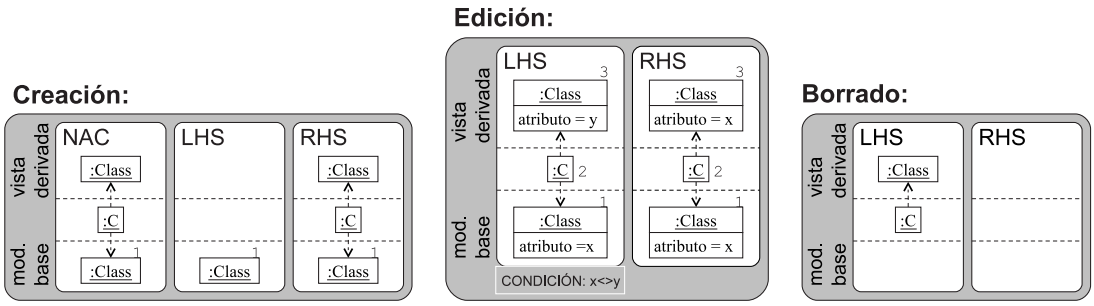


Figura 4.28: *Reglas básicas derivadas desde un patrón visual de consulta*

Adicionalmente, las reglas de C^i se extienden con condiciones de aplicación que tienen en cuenta las restricciones especificadas en el patrón visual de consulta. Tal como muestra la figura 4.29, por cada restricción negativa se añade una NAC a la regla de creación del tipo, y para el conjunto de restricciones positivas definidas sobre el tipo se crea una única PAC donde las restricciones son los grafos consecuencia. En la figura, la regla de creación básica para el tipo *Class* se modifica para incluir la NAC_{N1} derivada de la restricción negativa N_1, y la PAC $X_{Class} \rightarrow Y_{P1}$ derivada de la restricción positiva P_1. En el caso de aristas, las reglas de creación se modifican del mismo modo para incluir las condiciones de aplicación derivadas de restricciones sobre la arista, pero además también se añaden las condiciones derivadas de restricciones sobre sus nodos origen y destino. La razón es que para añadir una arista a la vista también los nodos en los que incide deben ser válidos. En el ejemplo, la regla de creación de la relación de herencia no añadiría condiciones de aplicación derivadas de restricciones sobre la relación, pero sí otras derivadas de las restricciones sobre la clase que actúa como origen y destino de la misma.

Finalmente, por cada conjunto de restricciones positivas sobre un tipo se crea una regla de borrado adicional en D^i donde las restricciones son NACs, tal y como muestra la regla derecha en la figura 4.30. La regla se aplica a un elemento si éste no cumple ninguna de las NACs, lo que significa que no satisface ninguna de las restricciones positivas P^i impuestas por el patrón. Por el contrario, a partir de cada restricción negativa se crea una nueva regla de borrado donde la restricción es una PAC, tal y como muestra la regla izquierda de la misma figura. En este caso se crea una regla por cada restricción, independientemente del tipo al que se aplica. Esto es, un elemento tiene que borrarse si cumple alguna de las

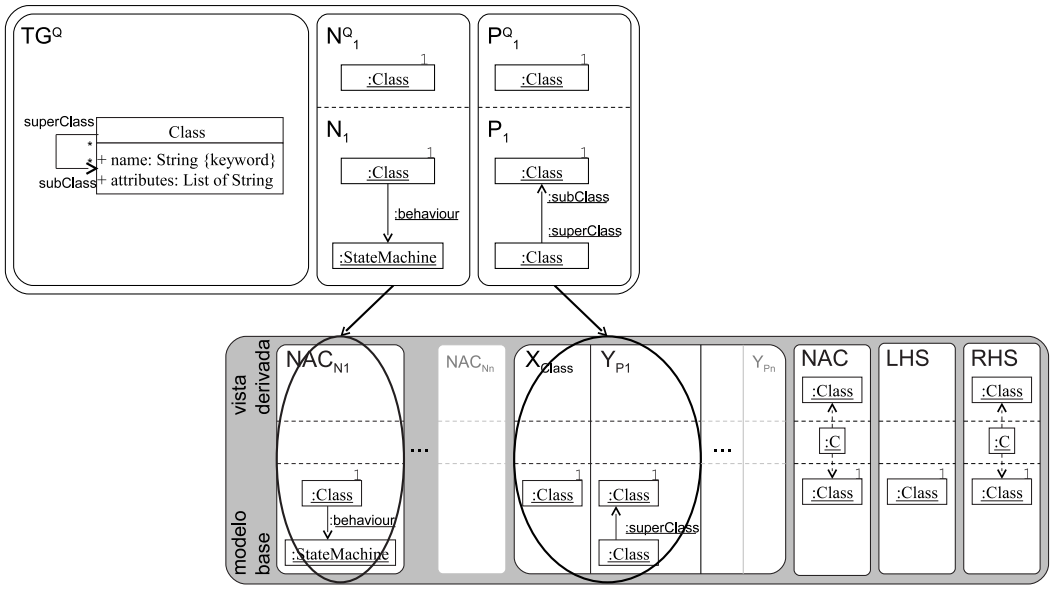


Figura 4.29: *Extensión de las reglas de creación con restricciones*

restricciones negativas N^i del patrón, razón por la cual se genera una regla de borrado por cada una de ellas. Obsérvese que, en el caso de aristas, las reglas de borrado que se generan tienen que considerar no sólo las restricciones impuestas sobre la arista, sino también las definidas para sus nodos origen y destino. Esto es así porque las aristas deben borrarse de una vista derivada si no cumplen las restricciones impuestas sobre ellas, o si inciden en un nodo que no cumple las suyas. Por tanto, aunque en el patrón del ejemplo no hay restricciones sobre aristas, se generarían tres reglas adicionales (dos derivadas de N_1 y una derivada de N_2) para el borrado de la relación de herencia cuando uno de sus extremos deja de ser válido.

Nótese que las vistas derivadas (y dirigidas a audiencia) pueden tener como modelo base cualquier vista del sistema o el repositorio. Si cualquier vista del sistema cambia, los cambios se propagan por medio de los TGTSs de consistencia al resto del sistema. Esto incluye al modelo base de la vista derivada, desde donde los cambios se propagan a la vista derivada mediante el TGTS generado a partir del patrón visual de consulta.

Por último, los TGTSs que se generan a partir de un patrón visual de consulta terminan y son confluentes. La demostración de terminación se sigue de los siguientes hechos:

- las reglas de creación contienen una NAC que es igual a la RHS,
- la condición de las reglas de edición deja de ser cierta tras su aplicación,
- las reglas de borrado eliminan elementos necesarios para su aplicación,

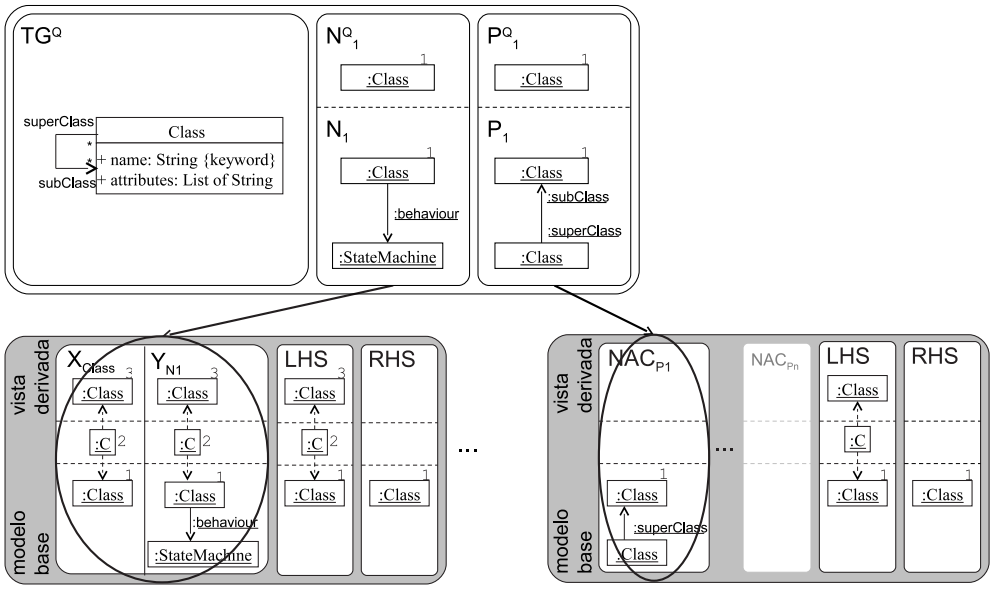


Figura 4.30: *Reglas de borrado adicionales derivadas de las restricciones*

- no puede darse un bucle en el que las reglas de creación borren lo que las de borrado eliminan, y viceversa, ya que por cada NAC de las reglas de creación hay una regla de borrado que tiene la NAC como PAC, de tal modo que si la regla de creación es aplicable las de borrado no pueden serlo simultáneamente. Del mismo modo, existe una regla de borrado que tiene como NACs las PACs de la regla de creación. Como las reglas no modifican los elementos que pertenecen a las condiciones de aplicación, la ejecución termina.

Respecto a la confluencia del TGTS, se debe a que las condiciones necesarias para aplicar las reglas siempre se evalúan sobre el modelo base, mientras que las acciones de las reglas tienen lugar sobre la vista derivada. Por esta razón no se producen conflictos entre las reglas o, si se producen, el resultado es confluente.

4.1.5. Implementación de la propuesta

Con el objetivo de evaluar empíricamente la factibilidad del marco propuesto en este capítulo, se ha construido una herramienta para especificar LVDEs multi-vista, lo que incluye la definición de sus distintos puntos de vista, vistas semánticas (TGTSs y patrones triples de anotación), llamadas a métodos de análisis y vistas dirigidas a audiencia. A partir de tal definición la herramienta genera un entorno visual para el lenguaje que permite crear vistas de un sistema que son conformes a alguno de los puntos de vista del lenguaje (consistencia intra-diagrama), e internamente proporciona mecanismos para la consistencia sintáctica y semántica entre vistas (consistencia inter-diagrama).

Para construir la herramienta se utilizó AToM^3 [51], un entorno de meta-modelado que permite generar entornos para LVDEs (no multi-vista) a partir de un meta-modelo de su sintaxis abstracta, y asociando una apariencia visual a cada uno de los elementos de dicha sintaxis. En concreto, se diseñó un LVDE que permite especificar entornos multi-vista según el marco propuesto, y a continuación se usó AToM^3 para generar una herramienta para el mismo. El lenguaje diseñado complementa el meta-modelo global de un LVDE multi-vista permitiendo definir sus puntos de vista, vistas semánticas y vistas dirigidas a audiencia. Su meta-modelo se muestra en la figura 4.31. La clase *MV-Environment* representa el entorno multi-vista, y sus tres atributos definen la política de consistencia sintáctica a implementar en el mismo: si realiza un borrado en cascada o conservativo, si hay creación inteligente, o si permite el cambio de identificador, respectivamente[3]. Además, un entorno multi-vista define tres tipos de vista: puntos de vista (clase *Viewpoint*), vistas semánticas (clase *SemanticView*) y vistas dirigidas a audiencia (clase *AudienceOrientedView*). No se han incluido vistas derivadas ya que éstas no forman parte de la definición de un LVDE multi-vista, sino que son el resultado de una consulta sobre un modelo construido posteriormente con dicho lenguaje.

Un punto de vista tiene un meta-modelo, una lista de propiedades (por ejemplo su autor, nombre, etc.), una apariencia gráfica y una cardinalidad que especifica el número mínimo y máximo de vistas del sistema que se pueden definir en un sistema para el punto de vista. El meta-modelo debe ser un subconjunto del meta-modelo global del LVDE multi-vista al que pertenece el punto de vista. Por otro lado, los atributos de cardinalidad son útiles, por ejemplo, para especificar que cierto tipo de vista es obligatorio en un lenguaje. Los puntos de vista definen relaciones de consistencia (asociación *view_consistency*) con TGTSs que implementan los mecanismos de consistencia sintáctica entre las vistas de un sistema. El repositorio también está definido según un punto de vista (no modificable) cuyo meta-modelo coincide con el del LVDE completo. De este modo los TGTSs para la consistencia sintáctica presentados en el capítulo se pueden especificar como asociaciones

[3]En la implementación realizada en AToM^3, esos atributos se codificaron como atributos del meta-modelo en vez de como una clase del mismo. Conceptualmente, ambas opciones son equivalentes.

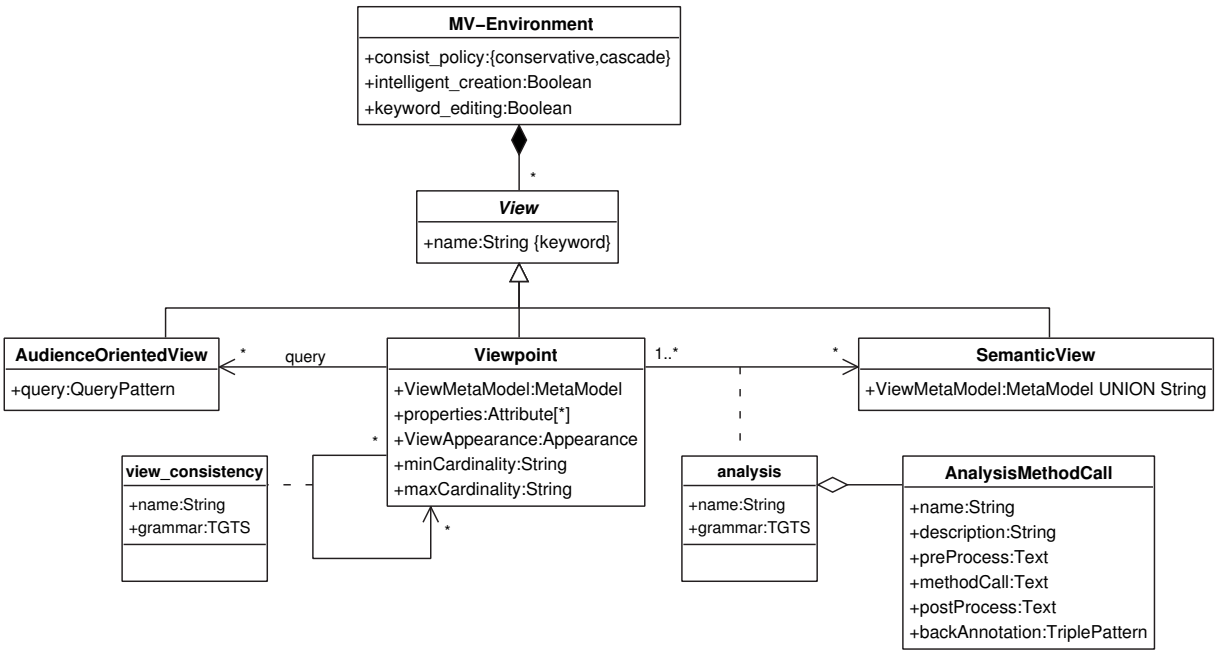


Figura 4.31: *Meta-modelo con los tipos de vista para la definición de un LVDE multi-vista*

view_consistency derivadas automáticamente a partir de la definición de los distintos puntos de vista.

Las vistas semánticas se definen especificando explícitamente su meta-modelo o dando el nombre de uno existente. Cada punto de vista (incluido el repositorio) puede definir un conjunto de vistas semánticas mediante relaciones de análisis que tienen un nombre, un TGTS para transformar las vistas del sistema conformes al punto de vista al dominio semántico, y un conjunto de métodos de análisis que evalúan propiedades de ese punto de vista en el dominio semántico. Cada método de análisis (clase *AnalysisMethodCall*) tiene un nombre, una descripción, acciones de pre- y post-procesamiento que se ejecutan antes y después de la llamada a una función de análisis, respectivamente, así como un patrón triple para la anotación de resultados. En la implementación realizada, las acciones tienen que expresarse en Python por ser éste el lenguaje en que está construida la herramienta.

Por último, es posible derivar vistas orientadas a audiencia desde los puntos de vista mediante la definición de consultas (asociación *query*). Una vista orientada a audiencia contiene un patrón de consulta cuyo componente TG^Q debe ser subconjunto del meta-modelo del punto de vista a partir del que se genera. Una vista orientada a audiencia no es derivable de una vista semántica, ya que el objetivo de esta última es proporcionar mecanismos de análisis transparentes al usuario (es decir, la transformación al dominio semántico y posterior anotación de resultados se realiza de manera oculta sin intervención por parte del usuario).

Respecto a la sintaxis concreta de los elementos del meta-modelo, tanto los puntos de vista como las vistas semánticas se representan mediante rectángulos amarillos con el nombre de la vista dentro, aunque los primeros tienen un tono más oscuro. En cambio las vistas orientadas a audiencia tienen la apariencia visual de un óvalo con el nombre de la vista en su interior. Relaciones de consistencia, relaciones de análisis y consultas se representan mediante flechas, en el último caso discontinuas. La figura 4.32 muestra un ejemplo de especificación de un entorno multi-vista utilizando la sintaxis concreta. El modelo contiene la definición de cuatro puntos de vista, y uno de ellos (`repository_DigitalLibrary`) define una vista semántica y una vista dirigida a audiencia.

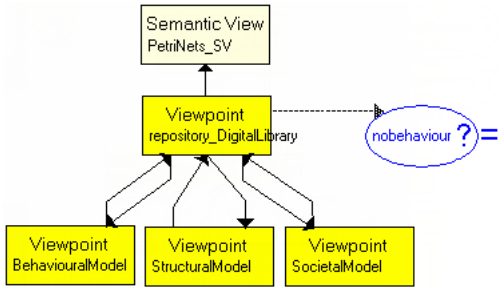


Figura 4.32: *Ejemplo de definición de lenguaje multi-vista utilizando la sintaxis concreta*

A partir del meta-modelo presentado y de la definición de su sintaxis concreta, AToM^3 generó automáticamente un entorno que permite especificar LVDEs multi-vista. Este entorno se completó con código escrito manualmente para la creación automática de relaciones de consistencia entre puntos de vista, entre otras características. Finalmente el entorno se integró en la interfaz de AToM^3 de tal manera que, una vez definido el meta-modelo completo y la sintaxis concreta de un LVDE multi-vista, se puede abrir el nuevo entorno para definir sus puntos de vista, vistas semánticas y vistas dirigidas a audiencia. La herramienta final se muestra en la ventana "1" de la figura 4.33, donde se han definido un conjunto de puntos de vista, una vista semántica y una vista dirigida a audiencia para un LVDE. Por defecto el entorno siempre incluye un punto de vista que no puede modificarse cuyo nombre comienza con `repository_` y que contiene el meta-modelo del LVDE completo. El diseñador del LVDE puede definir otros puntos de vista adicionales y editar sus atributos, tal y como muestran las ventanas "2" y "3" para la edición de los atributos y del meta-modelo del punto de vista llamado `StructuralModel`, respectivamente. Al crear un nuevo punto de vista la herramienta le asigna el meta-modelo del LVDE completo, el cual se puede modificar posteriormente para eliminar clases, relaciones y atributos, así como añadir restricciones (es decir, los puntos de vista deben ser subconjuntos del meta-modelo completo). También se puede asignar una sintaxis concreta distinta de la definida originalmente para los elementos del LVDE, de tal modo que un mismo elemento se visualice de manera distinta dependiendo del tipo de diagrama donde se esté mostrando.

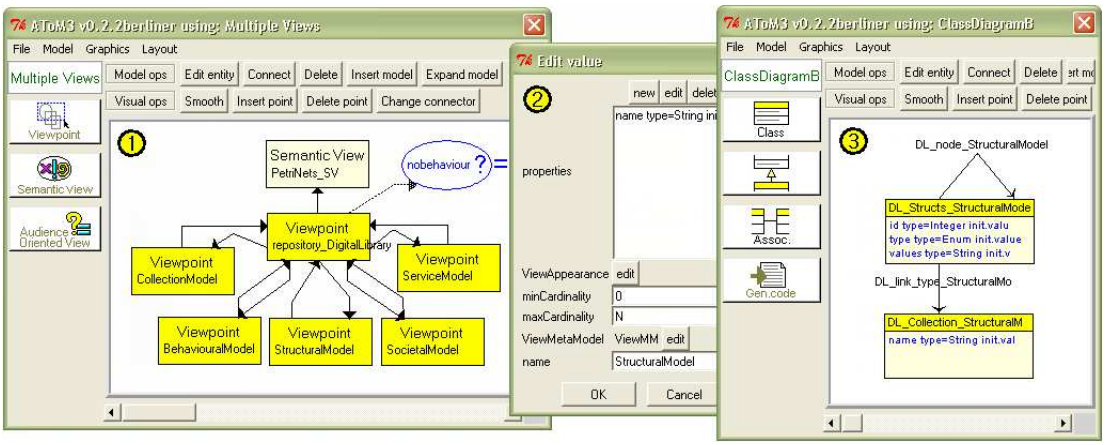


Figura 4.33: *Herramienta de soporte para la especificación de LVDEs multi-vista*

La herramienta genera automáticamente una relación de consistencia desde cada punto de vista al repositorio que contiene el TGTS para crear el repositorio a partir de las vistas del sistema, y una relación desde el repositorio a cada punto de vista que contiene el TGTS para propagar cambios. Los TGTSs generados dependen de la política de consistencia sintáctica seleccionada, pudiendo elegir entre borrado conservativo o en cascada, así como activar o no la creación inteligente de elementos o la edición de identificador. Posteriormente se pueden modificar para añadir nuevas reglas con restricciones sobre la semántica estática del lenguaje, así como para modificar el comportamiento generado por defecto si es necesario. Como las reglas son visuales y utilizan la sintaxis concreta del LVDE, no es necesario conocer el API de ningún lenguaje de programación específico para poder modificarlas. Además, el hecho de que se utilice la sintaxis concreta del lenguaje para su construcción mejora su usabilidad. Por ejemplo, la figura 4.34 muestra una de las reglas triples de consistencia generadas automáticamente a partir de la definición de los puntos de vista. En concreto la regla se encarga de la creación en el repositorio de nuevos elementos del tipo usado en la regla.

Es posible definir nuevas relaciones de consistencia entre los puntos de vista. Cuando en el entorno generado a partir de esta descripción se modifique una vista del sistema, se ejecutarán los TGTSs de las relaciones de consistencia de salida de su punto de vista; si esto provoca la modificación de otra vista, el proceso se repetirá para propagar los cambios.

La herramienta también permite definir vistas semánticas a partir de los puntos de vista, tal como muestra la figura 4.33 para la vista semántica `PetriNets_SV` desde el punto de vista `repository_DigitalLibrary`. Para ello se debe especificar un TGTS y, por cada propiedad a verificar en el dominio semántico, un método de análisis. La figura 4.35 muestra un ejemplo de especificación de regla triple de transformación en AToM^3. Los componentes de la regla triple son grafos triples formados por un grafo conforme al punto de vista (el

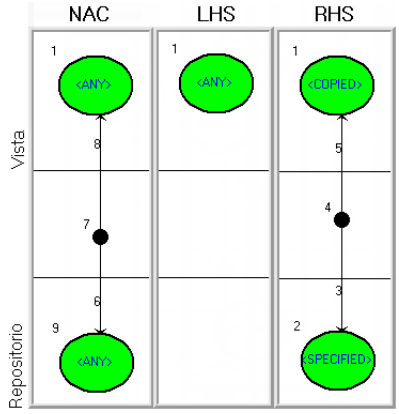


Figura 4.34: *Regla triple generada automáticamente para la consistencia sintáctica entre vistas del sistema*

repositorio en este caso), otro conforme al dominio semántico, y un grafo correspondencia.

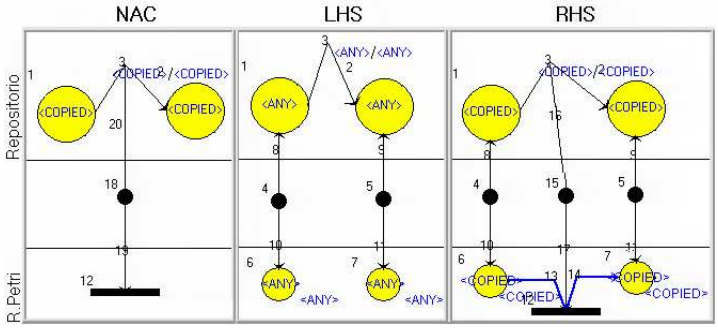


Figura 4.35: *Regla triple para la transformación de un punto de vista a un dominio semántico*

En el entorno generado a partir de esta descripción se creará un botón por cada método de análisis que permite evaluar la propiedad correspondiente sobre los modelos conformes al punto de vista desde el que se genera la vista semántica (sobre el repositorio en el ejemplo). El resultado se mostrará de acuerdo al mecanismo de anotación definido. Si se especifica mediante un patrón triple de anotación como el de la figura 4.36, los elementos de salida (etiquetados "1", "2" y "3" en el patrón de ejemplo) se mostrarán resaltados en el modelo original y resumidos en un cuadro de diálogo que, además, permite navegar entre distintas soluciones de un mismo análisis mostrando una cada vez en caso de ser necesario. Además, en el entorno resultante se genera un botón adicional que muestra el grafo destino resultante de la ejecución del TGTS. Esto es útil si el dominio semántico proporciona herramientas de simulación o animación.

Finalmente, para definir vistas dirigidas a audiencia en la nueva herramienta se debe indicar tanto el patrón de consulta utilizado para su obtención, como el punto de vista sobre el que realizar la consulta. La figura 4.33 muestra una vista dirigida a audiencia llamada

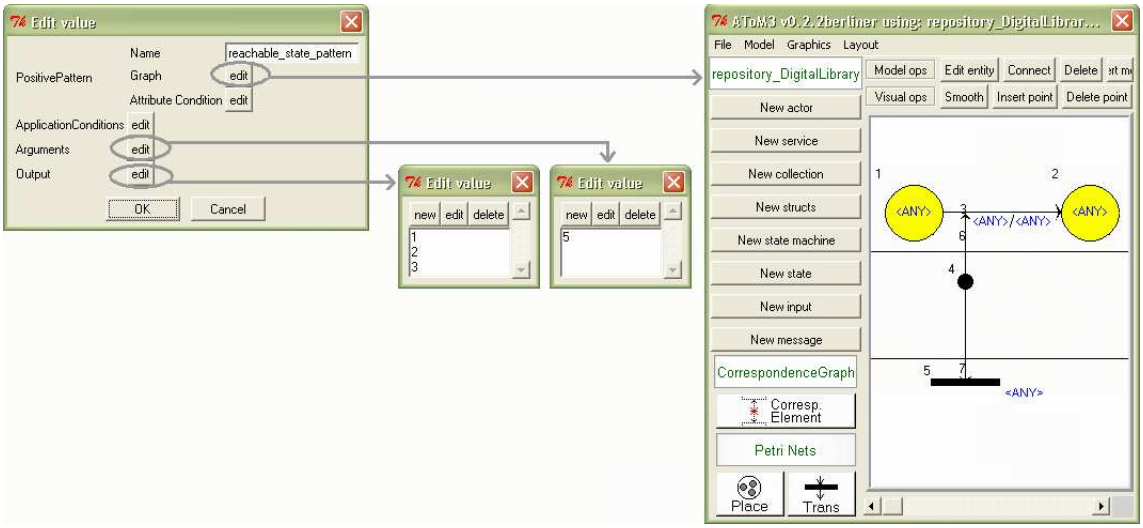


Figura 4.36: *Especificación de un patrón triple de anotación en la nueva herramienta*

`nobehaviour`, definida para el punto de vista `repository_DigitalLibrary`, y cuyo patrón de consulta se muestra en la figura 4.37. Dicho patrón consta de un meta-modelo que debe ser una restricción del punto de vista consultado (esquina superior derecha), y de restricciones sobre sus elementos (esquina inferior derecha). En el entorno generado se creará un botón por cada vista dirigida a audiencia que permite evaluar el patrón de consulta sobre los modelos del punto de vista para el que se define.

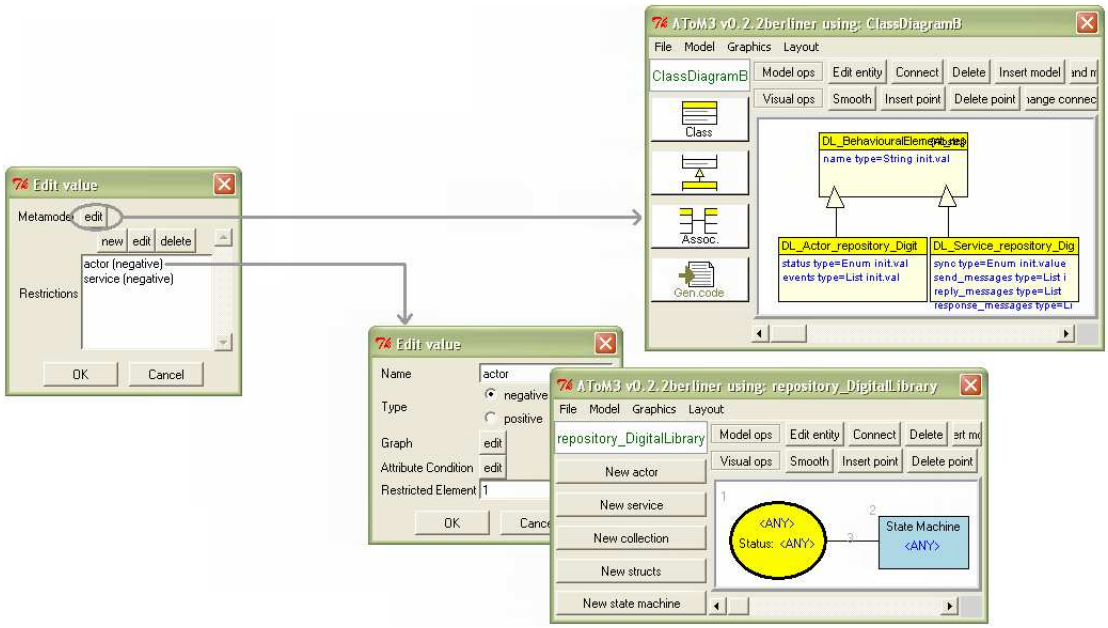


Figura 4.37: *Especificación de un patrón de consulta en la nueva herramienta*

Como ejemplo, la figura 4.38 muestra el entorno visual generado para un LVDE multi-vista a partir de la especificación anterior. La ventana del fondo permite crear y editar vistas del sistema, y la ventana en primer plano corresponde a la edición de una de esas vistas. Internamente, la modificación de una vista dispara la ejecución de los TGTSs incluidos en las relaciones de consistencia definidas para el lenguaje. De este modo los cambios realizados en una vista se propagan al repositorio, y desde allí al resto de vistas del sistema.

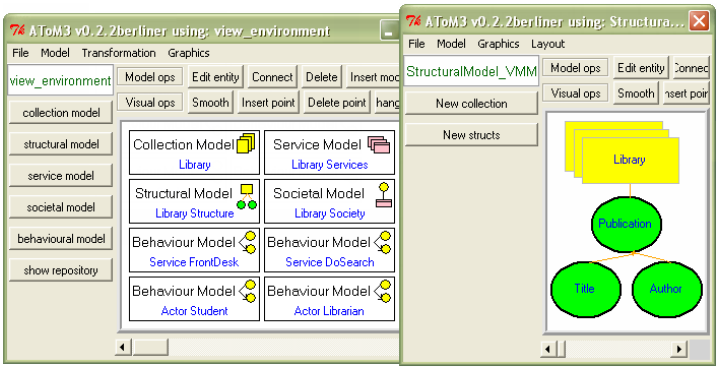


Figura 4.38: *Entorno visual multi-vista generado*

El entorno incluye un botón para mostrar el repositorio construido a partir de las vistas del sistema. Aunque el repositorio se puede modificar, por ejemplo para llevar a cabo una simulación, los cambios no se guardan ni se propagan al resto de vistas del sistema.

La figura 4.39 muestra la interfaz del repositorio generada para el ejemplo. En la interfaz, además de los botones para la creación de elementos del lenguaje, se genera un botón por cada método de análisis especificado sobre las vistas semánticas del repositorio, así como uno por cada vista orientada a audiencia definida sobre él (recuérdese que la definición del lenguaje incluía una vista semántica y una orientada a audiencia para el repositorio). Para verificar una propiedad sólo hay que pulsar el botón correspondiente. Internamente, la herramienta realiza la transformación al dominio semántico, llama a la función de análisis correspondiente y devuelve el resultado de acuerdo al mecanismo de anotación, escondiendo al usuario todo el proceso de análisis. En la figura se muestra el resultado de verificar una propiedad sobre el repositorio. Como en este caso el análisis obtuvo varios resultados, la ventana permite navegar de un resultado a otro por medio de los botones inferiores. Cada solución individual se muestra resaltada en el modelo y resumida en la ventana de diálogo. De este modo, el resultado de los análisis se muestra en términos de la notación original al usuario, quien no necesita tener conocimiento del método formal subyacente utilizado para la verificación. Igualmente, para obtener una vista orientada a audiencia basta pulsar el botón correspondiente y el modelo resultante de la consulta se muestra en una nueva ventana.

El capítulo 5 presenta una descripción más detallada del ejemplo usado en esta sección

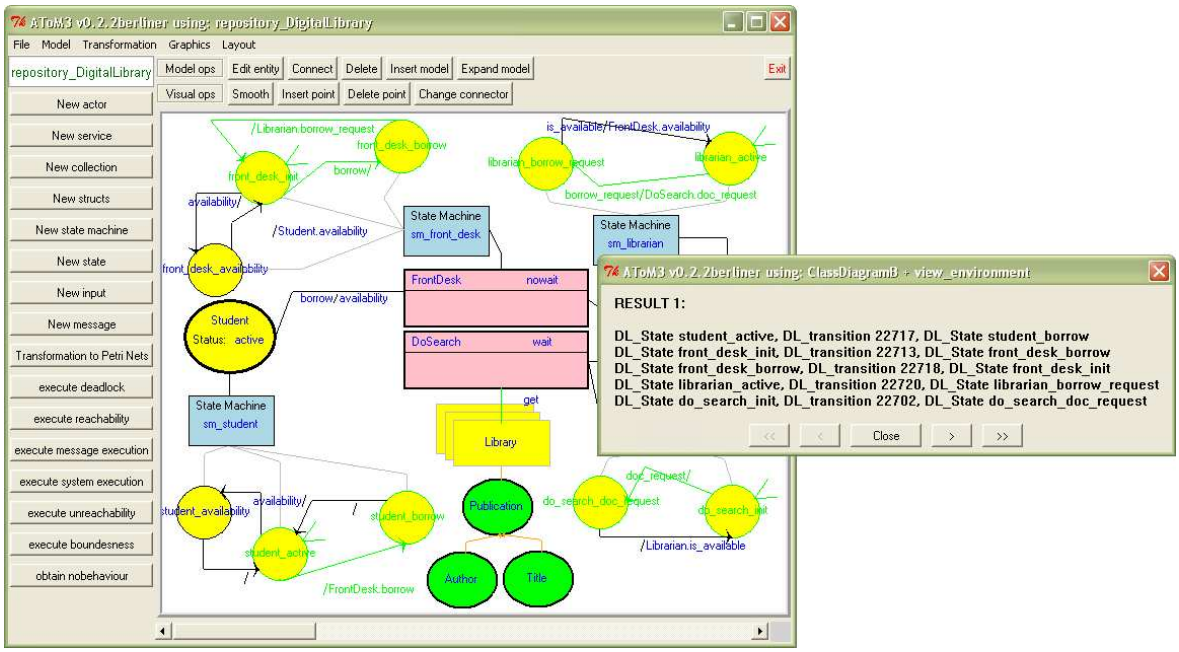


Figura 4.39: *Ejecución de un método de análisis sobre el repositorio*

para ilustrar el uso de la herramienta construida, así como otros ejemplos adicionales.

4.2. Soporte para medidas y rediseños

Esta sección presenta un marco de especificación de medidas y rediseños para LVDEs que ayuden a la obtención de modelos que cumplan requisitos implícitos de calidad, como facilidad de mantenimiento, usabilidad, etc. El objetivo del marco es doble: (i) facilitar la definición de medidas y rediseños para cualquier LVDE; y (ii) permitir la generación automática de herramientas de medición y rediseño desde una descripción de alto nivel, reduciendo o eliminando la necesidad de codificarlas. De esta manera, el conocimiento de los expertos en calidad dentro de un dominio se plasma en forma de modelos que se utilizan para generar código. Además, este conocimiento no permanece estático en forma de guías o documentación, sino que los diseñadores lo utilizan activamente en la fase de modelado.

Para cubrir el primero de los objetivos se ha elaborado un LVDE denominado SLAMMER (siglas del inglés *Specification LAnguage for Modelling MEasures and Redesigns*). SLAMMER se ha especificado mediante un meta-modelo que contiene generalizaciones de algunas métricas de producto y rediseños comúnmente utilizados, y que pueden configurarse mediante el uso de patrones visuales que indican cómo ciertos atributos relevantes para la métrica o el rediseño se expresan en una notación en particular. Además, el lenguaje también permite expresar de manera procedimental o mediante reglas de transformación de grafos otros rediseños adicionales. Finalmente, es posible expresar mediante indicadores qué valores de una métrica son inusuales, y asociar a dichos valores la ejecución de ciertas acciones orientadas a mejorar el resultado de la medición.

Para conseguir el segundo objetivo, se ha construido una herramienta de soporte para SLAMMER con un generador de código integrado que produce automáticamente herramientas para la medición y rediseño de modelos de dominio a partir de modelos SLAMMER. Esta herramienta se ha integrado en el entorno de meta-modelado AToM^3. De este modo, al describir un LVDE es posible establecer las métricas y rediseños propios del lenguaje, los cuales estarán disponibles en el entorno visual generado para el mismo.

La sección está dividida en cuatro apartados. Los dos primeros presentan el meta-modelo de SLAMMER, el primero la parte que concierne a la definición de medidas y el segundo la que concierne a la definición de acciones y rediseños. El segundo apartado también incluye una discusión sobre la integración de mecanismos de consistencia en un proceso de rediseño que incluya distintos modelos. El tercer apartado dota de una sintaxis concreta al meta-modelo de SLAMMER. Para finalizar, el cuarto apartado presenta la implementación de la herramienta de soporte para el lenguaje.

4.2.1. Medidas

Una medida o métrica de una (o varias) entidades se define mediante un método de medición y una escala [97]. Por tanto, para definir adecuadamente una medida se debe especificar la siguiente información:

- el dominio o conjunto de entidades que se quiere caracterizar;
- el método de medición, que consiste en una función que utilizará el resultado de otras mediciones en el caso de medidas indirectas;
- los atributos relevantes para el método de medición;
- la escala, que es el rango de valores posibles para el resultado de la medición y puede ser nominal, ordinal, absoluta, intervalo o ratio [77];
- una unidad de medición si la escala es de tipo intervalo o ratio (por ejemplo número de clases);
- adicionalmente, puede incluir indicadores que apunten a valores dentro de su escala que son inusuales, inadecuados o especiales por algún motivo.

Si tomamos una medida así definida e intentamos aplicarla a distintos dominios, se observa que parte de esa información es independiente de los conceptos del dominio particular ya que su definición no varía, independientemente de dónde se aplica. Ese es el caso del método de medición y de la escala. Por el contrario, hay otra parte de esa información que sí depende del dominio y del LVDE sobre los que se aplica, y por tanto debe especificarse para cada caso concreto. Es lo que ocurre con el dominio, los atributos relevantes para el método de medición, las unidades y los indicadores.

Por ejemplo, supongamos que quiere medirse el Número de Hijos directos (*Number of Children*, NOC) [191] en los dos primeros modelos de la figura 4.40, los cuales corresponden a un diagrama de clases y a una jerarquía de roles de usuario respectivamente. El método de medición, que es el mismo en ambos casos, cuenta el número de entidades subordinadas a una entidad dada dentro de la jerarquía. La escala también coincide: valores en el rango $[0, N]$ (donde N es un número natural arbitrario). Sin embargo, el dominio en el primer caso es el conjunto de clases que forman el modelo (es decir, la medición se realiza para cada clase en el modelo) mientras que en el segundo es el conjunto de roles. El atributo relevante para el método de medición en este caso es cómo se expresa la relación de herencia en el lenguaje. Esta relación se muestra resaltada en los modelos de la figura, y como puede verse es distinta. Finalmente, la unidad de medición es "clases" en el primer caso y "roles" en el segundo. Incluso, se puede ir un paso más lejos y aplicar esta medida a lenguajes que no incorporan el concepto de herencia, como es el caso del diagrama de navegación web

que se muestra como tercer ejemplo en la figura 4.40. En este caso el NOC se interpretaría como el número de nodos de información accesibles desde un nodo dado, donde el atributo relevante para el método de medición sería el concepto de enlace entre nodos. Es más, en este lenguaje, si un nodo tuviese un NOC igual a 0 significaría que el grafo de navegación no es totalmente conexo, lo cual suele ser indicativo de un mal diseño [26]. En cambio, en los dos primero lenguajes los elementos medidos pueden tener un NOC igual a 0.

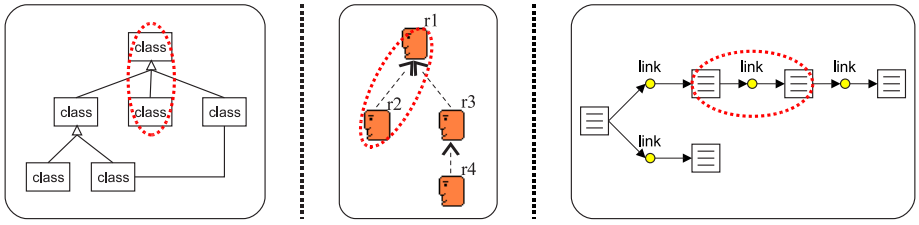


Figura 4.40: *Atributo relevante para el mismo método de medición, en distintos lenguajes*

En SLAMMER se ha utilizado el hecho de que hay información que cambia y otra que permanece inmutable al definir la misma medida para distintos dominios, para especificar un conjunto de medidas genéricas que esconden el método de medición y que pueden ser configuradas para distintos LVDEs dando sólo la información específica de dominio en cada caso. La figura 4.41 muestra la parte del meta-modelo de SLAMMER que formaliza estos conceptos. Para su definición se han tenido en cuenta trabajos de diversos autores enfocados a la definición de ontologías de medición software [77, 128], así como al estándar internacional para la calidad del software ISO 15939 [97]. Sin embargo, ya que el lenguaje se quiere utilizar como notación de alto nivel para la especificación y posterior generación de herramientas de medición, en el meta-modelo no se han incluido conceptos que no tuvieran un significado operacional. Esta es la razón de que términos tales como "necesidad de información" o "modelo de calidad" que aparecían en los trabajos anteriormente citados no hayan sido incluidos en el meta-modelo de SLAMMER.

El concepto medida está representado en el meta-modelo SLAMMER mediante la clase abstracta *Measure*, que actúa como clase base de cualquier tipo de medida y define los atributos comunes a todas ellas (véase la figura 4.41). Tiene un nombre que identifica unívocamente la medida (atributo *name*) y un objetivo (atributo *goal*). Su dominio se especifica mediante una lista ordenada de tipos (atributo *domain*), y se construye como todas las combinaciones posibles de elementos de esos tipos en un modelo dado. El atributo *subtypeMatching* permite incluir en la construcción del dominio no sólo los elementos de los tipos especificados en el atributo *domain*, sino también las de sus subtipos. De este modo es posible definir medidas reutilizables que se definen para un tipo general y se aplican a todos sus subtipos. Además, cada medida se mide según una escala (atributo *scale*) y utilizando una unidad de medición (atributo *unit*) que se definen mediante una cadena de texto. Para la especificación de medidas indirectas que se calculan en función de otras

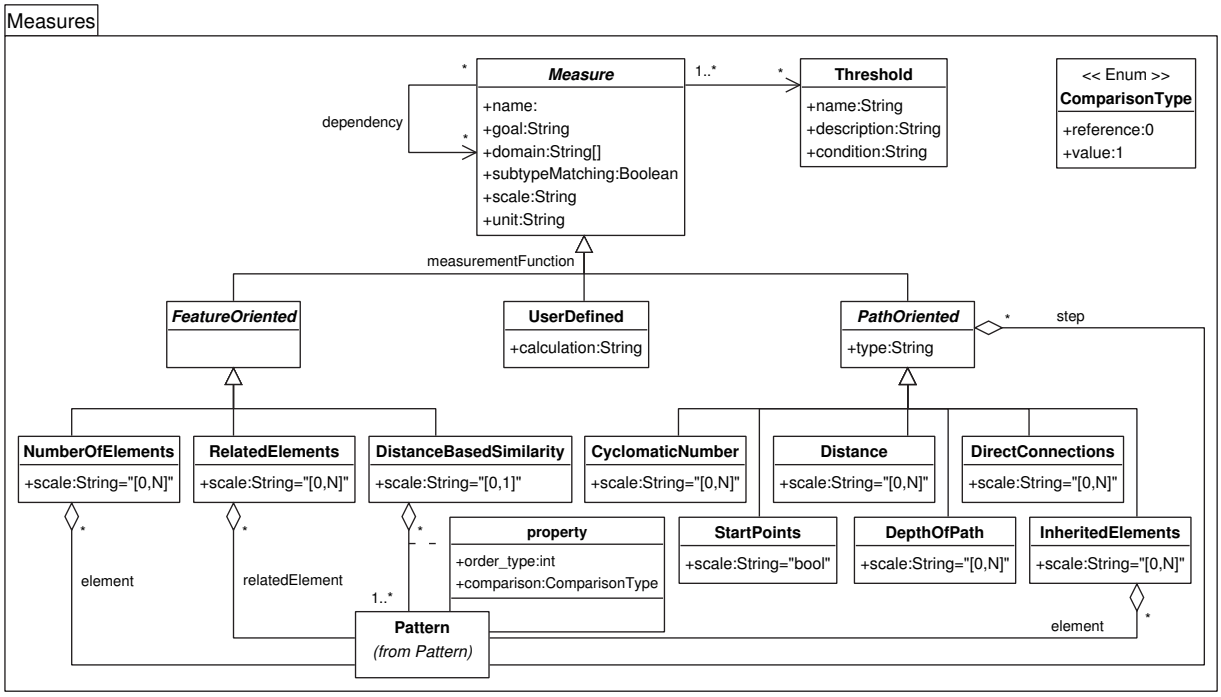


Figura 4.41: *Meta-modelo de SLAMMER (paquete para la definición de medidas)(i)*

medidas (que a su vez pueden ser directas o indirectas), el meta-modelo define la relación *dependency*. Si una medida "depende" de otra significa que la primera utiliza el resultado de la medición de la segunda para calcular su resultado. De este modo una medida puede ser reutilizada y compuesta para construir otras más complejas. El meta-modelo incluye una restricción que prohíbe la definición de ciclos recursivos de dependencias. Finalmente, una medida puede definir cero, uno o más indicadores de valores extremos dentro de la escala mediante la especificación de valores umbral (clase *Threshold*). Cada umbral tiene un nombre, una descripción y una condición lógica que se evalúa sobre cada resultado de la medición para determinar si dicho resultado pertenece o no al umbral.

El meta-modelo de SLAMMER define un conjunto de medidas genéricas como clases concretas que heredan de la clase *Measure* todos sus atributos. Esas medidas se clasifican según dos características ortogonales, que son la funcionalidad de su método de medición y la dimensión de su dominio. Ambas clasificaciones se muestran en las figuras 4.41 y 4.42, respectivamente. Las figuras son dos puntos de vista del mismo meta-modelo, aunque se muestran por separado para facilitar su legibilidad. Esto es, las medidas que aparecen en ellas son las mismas, pero clasificadas desde una perspectiva distinta. El hecho de modelar explícitamente en el meta-modelo esa categorización de las medidas favorece la extensibilidad del enfoque. Desde un punto de vista práctico, si la herramienta de soporte para el enfoque propuesto se construye a partir de este meta-modelo explícito (utilizando

por ejemplo, una herramienta de meta-modelado) resultará más fácil extender y mantener el conjunto de medidas.

Tomando en consideración la función de medición, tal como muestra la figura 4.41, las medidas pueden ser:

- Orientadas a características (clase *FeatureOriented*): miden propiedades de objetos o grupos de objetos, de tal modo que es el objeto lo que se intenta caracterizar. Por ejemplo, el número de métodos de una clase o la similitud de dos clases son medidas orientadas a características.

- Orientadas a caminos (clase *PathOriented*): miden propiedades de caminos entre elementos del mismo tipo, de tal modo que es la estructura de los caminos lo que se intenta caracterizar. Por ejemplo, la herencia en lenguajes orientados a objetos puede considerarse un tipo de camino donde los nodos del camino son las clases, y el camino lo marcan las relaciones de herencia existentes entre ellas. En este tipo de medidas el atributo *type* recoge el tipo de los nodos del camino a medir, mientras que la relación que establece un paso en el camino se especifica mediante un patrón. Esto último permite utilizar como paso relaciones compuestas.

- Orientadas a usuario (clase *UserDefined*): en este tipo de medidas la función de medición la especifica el diseñador. Para ello la clase dispone de un atributo denominado *calculation* para incluir el código que calcula el resultado de la medición para un valor del dominio. El código se encapsula en un método que recibe como parámetros un valor del dominio, y devuelve un valor como resultado del cálculo. Las medidas orientadas a usuario se utilizan para especificar medidas que no pueden ser expresadas con el conjunto de medidas genéricas que SLAMMER proporciona. Esto ocurre con aquellas medidas que son altamente dependientes del dominio y para las que, por tanto, no se ha incluido soporte en el meta-modelo de SLAMMER. Estas medidas también permiten recibir como entrada los resultados de otras mediciones y realizar operaciones con ellos.

Como muestra la figura 4.42[4], según la dimensión del dominio las medidas pueden ser:

- Orientadas a modelo (clase *ModelOriented*): miden propiedades del modelo, como por ejemplo el tamaño. En estas medidas el dominio está formado por un único valor, que es el modelo completo. Por esa razón su atributo *domain* es NULL, ya que no es necesario especificarlo.

[4]Nótese que el meta-modelo de la figura 4.42 es un punto de vista del meta-modelo de la figura 4.41, y por tanto contiene las mismas clases que éste aunque no se muestren de manera explícita.

- Orientadas a elemento (clase *ElementOriented*): miden propiedades de los elementos, como por ejemplo el número de métodos de una clase. En este caso el dominio de la medida está formado por los elementos de un tipo determinado, y por tanto basta con un *String* para especificar dicho tipo.
- Orientadas a grupo (clase *GroupOriented*): miden propiedades de grupos de elementos, como su acoplamiento o similitud.

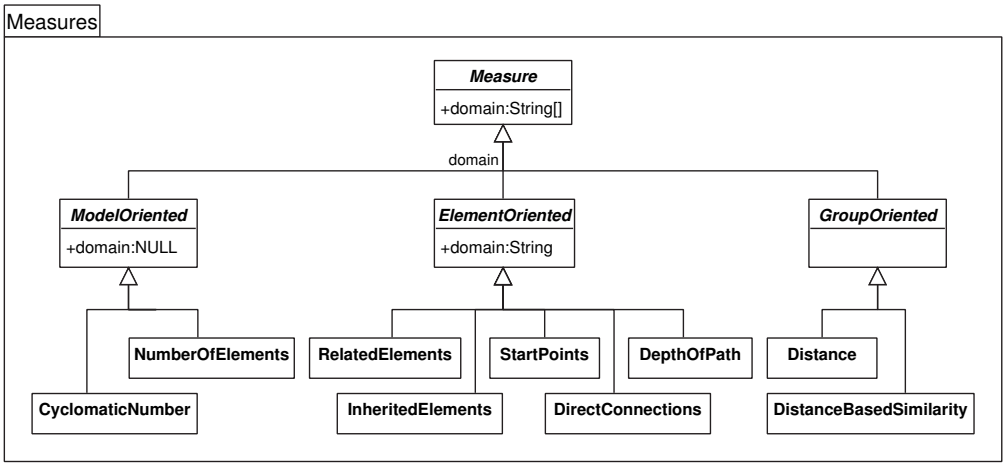


Figura 4.42: *Meta-modelo de SLAMMER (paquete para la definición de medidas)(ii)*

En total, SLAMMER define 9 medidas que son generalizaciones o abstracciones de algunas de las métricas más utilizadas en ingeniería del software [68, 129, 167, 187]. La tabla 4.1 las recoge y clasifica según su función de medición y el tamaño de su dominio. Su configuración para un LVDE concreto se realiza mediante el uso de patrones visuales (clase *Pattern* en el meta-modelo) que identifican cómo ciertos atributos relevantes para el cálculo de la métrica se expresan en el lenguaje. A grandes rasgos, un patrón visual es un grafo más un conjunto de restricciones que especifican el tipo de estructuras que queremos encontrar en un modelo. Posteriormente se explicará de manera más detallada cómo se utilizan los patrones dentro de SLAMMER.

A continuación se presentan cada una de las 9 medidas:

- *NumberOfElements*: calcula el número de elementos de cierto tipo existentes en un modelo. Esta medida es orientada a modelo porque calcula una propiedad del modelo y por tanto no es necesario especificar el dominio. El tipo de los elementos a contar se especifica mediante un patrón. Ya que los patrones son grafos con restricciones adicionales, es posible contar no sólo elementos de cierto tipo, sino también estructuras complejas formadas por grupos de elementos relacionados o con ciertas restricciones

	Orientadas a modelo	**Orientadas a elemento**	**Orientadas a grupo**
Orientadas a características	NumberOfElements	RelatedElements	DistanceBasedSimilarity
Orientadas a caminos	CyclomaticNumber	StartPoints DepthOfPath DirectConnections InheritedElement	Distance

Tabla 4.1: *Clasificación de medidas en SLAMMER*

(por ejemplo elementos de cierto tipo que no están relacionados con elementos de algún otro tipo).

- *RelatedElements*: cuenta cuántos elementos están relacionados con cierto tipo de elemento, el cual se especifica en el atributo *domain*. Esta medida es orientada a elemento, y por tanto se calcula para cada elemento del tipo especificado en el modelo dado. La relación entre los elementos se especifica mediante un patrón, lo que permite expresar relaciones compuestas.

- *DistanceBasedSimilarity*: evalúa el grado de similitud de un conjunto de entidades mediante el estudio de los atributos que comparten [167]. Sean x e y las entidades a comparar, y sea $b(\cdot)$ una función que devuelve el conjunto de atributos de una entidad. Entonces, la fórmula $dist(x, y) = 1 - \frac{|b(x) \cap b(y)|}{|b(x) \cup b(y)|}$ calcula la distancia entre las dos entidades, y puede tomar valores en el intervalo [0,1]. Cuanto mayor es el resultado de la medición, mayor es la distancia entre las entidades medidas y por tanto más diferentes son éstas entre sí. Los tipos de las entidades a comparar se dan mediante una lista en el atributo *domain*. Además, hay que especificar para cada tipo el conjunto de propiedades usadas para establecer la comparación. Cada propiedad se representa mediante un patrón (relación *property* en el meta-modelo), y debe darse si la comparación para la propiedad específica se hará por referencia o por valor. En la comparación por referencia dos objetos se consideran iguales si son el mismo objeto, mientras que en la comparación por valor dos objetos son iguales si todos sus campos tienen el mismo valor.

- *Distance*: tanto ésta, como las medidas que vienen a continuación, son orientadas a caminos. Para configurarlas hay que definir el tipo de nodo en el camino (atributo *type*) y el paso básico (mediante un patrón). Al utilizar un patrón para especificar la relación entre dos nodos del camino es posible especificar relaciones complejas formadas por varios nodos y enlaces. El dominio de cada medida se construye a partir del tipo de nodo y teniendo en cuenta la dimensión del dominio de la medida.

En total hay seis medidas orientadas a caminos. La medida *Distance* mide la longitud del camino más corto desde cada nodo del camino al resto. Esta medida es orientada a grupo, ya que se calcula para cada pareja de nodos (del tipo especificado) en el modelo dado. El resultado, por tanto, es una matriz donde la posición (i, j) es la longitud del camino desde el nodo i hasta el nodo j.

- *DirectConnections*: calcula el número de nodos a los que se puede llegar desde cada nodo del camino en un camino de longitud 1. Por ejemplo, puede utilizarse para medir el número de hijos directos en el caso de herencia.

- *StartPoints*: informa sobre qué nodos son inicio de un camino pero ningún otro camino pasa por él. Por ejemplo, las clases base en lenguajes orientados a objetos son los puntos de inicio de un camino jerárquico.

- *DepthOfPath*: da la distancia del camino más corto para alcanzar cualquier nodo desde el inicio de un camino (esto es, desde un *StartPoint*). Por ejemplo, en el caso de la herencia esa es la profundidad del árbol de herencia.

- *CyclomaticNumber*: cuenta el número de caminos circulares (ciclos) en un modelo dado.

- *InheritedElements*: esta última medida es aplicable básicamente a caminos con un significado de herencia. Mide el número de elementos de cierto tipo que son heredados a través de una jerarquía de herencia (por ejemplo, el número de métodos que una clase hereda de sus ancestros). En este caso, además del tipo de nodo y el paso básico en el camino, hay que especificar la relación entre el nodo (clase) y el elemento que se propaga (método) mediante un patrón.

Patrones visuales dentro del contexto de SLAMMER

Como ya se ha explicado, en SLAMMER se utilizan patrones visuales para especificar cómo ciertas propiedades se expresan en un LVDE determinado, y de ese modo poder configurar las medidas genéricas proporcionadas para el lenguaje específico.

Formalmente, la definición de patrón visual utilizada en SLAMMER coincide con la presentada en el apartado 4.1.3 para patrones triples de anotación, pero usando grafos tipados atribuidos (y morfismos correspondientes) en vez de grafos triples. Se remite al lector interesado a dicho apartado para consultar los detalles técnicos, de los que aquí sólo se da una definición informal. Así pues, en SLAMMER un patrón visual consta de un *grafo positivo* y su aplicación a un modelo da como resultado aquellos subgrafos del modelo que son isomorfos al grafo positivo. El patrón puede recibir como parámetro un conjunto de elementos del modelo al que se aplica para inicializar la búsqueda, y el resultado puede

filtrarse para obtener sólo una parte de los subgrafos obtenidos como resultado. También es posible restringir el número de subgrafos resultantes de la aplicación de un patrón mediante condiciones de aplicación [61] formadas por un grafo premisa y un conjunto de grafos consecuencia. Para que un subgrafo de un modelo sea un resultado válido de aplicar un patrón, debe ser isomorfo al grafo positivo del patrón y cumplir sus condiciones de aplicación.

Por ejemplo, la figura 4.43 muestra a la izquierda un patrón cuyo grafo positivo contiene un objeto *Rol* y otro *Nodo* unidos por una relación *PA*. El patrón tiene como parámetro el rol ya que el atributo *arguments* contiene el número "1", que es la etiqueta del rol en el grafo positivo. La aplicación de patrón sólo devolverá los objetos de tipo *Nodo* de cada subgrafo resultante ya que el atributo *output* contiene el número "3", que es la etiqueta del nodo en el grafo positivo. A la derecha, el patrón se aplica a un grafo G para el rol `r1`. En el paso (i) se inicializa la búsqueda para el rol recibido como parámetro. En el paso (ii) el subgrafo que contiene el rol se amplía para completar el grafo positivo. En el ejemplo se encuentran dos subgrafos: una que relaciona el rol `r1` con el nodo `n1`, y otra que lo relaciona con el nodo `n2`. En el paso (iii) se filtran los subgrafos de tal modo que sólo los elementos especificados como salida se devuelven como resultado. Por tanto, ya que el patrón especifica el elemento "3" como salida, sólo los nodos `n1` y `n2` se devuelven como resultado.

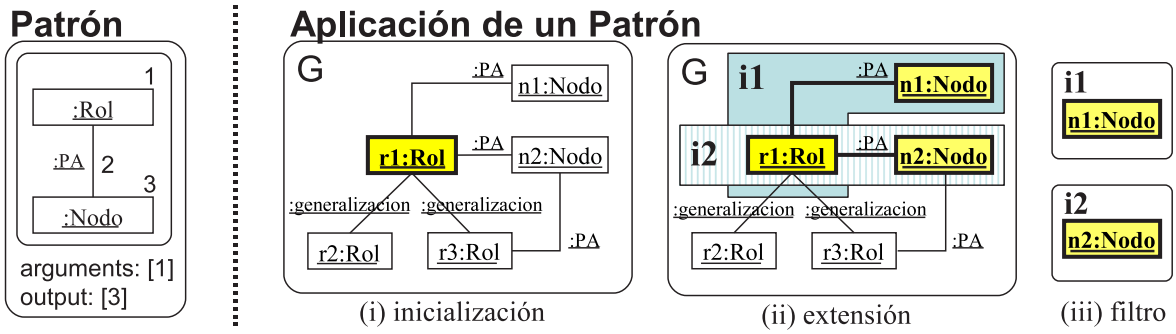


Figura 4.43: *Ejemplo de patrón visual y aplicación a un grafo*

En SLAMMER los patrones visuales se utilizan para especificar cómo los atributos relevantes para una medida se expresan en un LVDE dado. Los parámetros del patrón corresponden a un valor en el dominio de la medida, y su salida son los atributos que se quieren obtener para dicho valor del dominio. Por ejemplo, el patrón de la figura 4.43 se puede utilizar para configurar una medida de tipo *RelatedElements* para un LVDE que incluya los conceptos de rol, nodo y asignación de permisos. Como se recordará, esta medida cuenta el número de elementos relacionados con un tipo de elemento dado que se especifica en el atributo *domain*. El tipo, en este caso, sería "Rol". La medida así configurada contaría el número de nodos a los que puede acceder el rol recibido como parámetro, y que es dos

para el caso del rol `r1`. Debe tenerse en cuenta que la medida se realizaría para todos los posibles valores del dominio, esto es, para cada uno de los roles que contiene el grafo G.

El meta-modelo de SLAMMER incluye la definición de patrón visual que muestra la figura 4.44. De este modo, un patrón (clase *Pattern*) está formado por un grafo positivo y tiene como atributos sendas listas con las etiquetas de los elementos del grafo positivo que forman parte de la entrada y salida del patrón (atributos *arguments* y *output* respectivamente). El grafo positivo se modela mediante la clase *PatternGraph*, la cual contiene un nombre, un grafo y una condición sobre los atributos que se expresa en un lenguaje textual. Un patrón puede definir cero, una o más condiciones de aplicación. Cada condición de aplicación está formada por un grafo premisa y un conjunto de grafos consecuencia, cuya semántica fue expuesta previamente. Nótese que la clase *Pattern* definida en este meta-modelo es la misma que usa el meta-modelo de medidas de la figura 4.41.

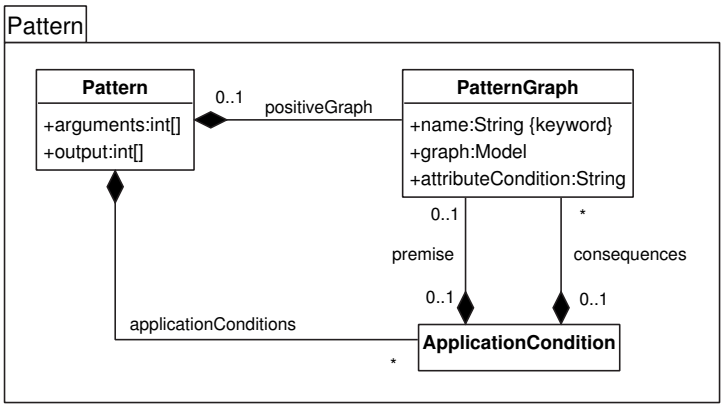


Figura 4.44: *Meta-modelo de SLAMMER (paquete para la definición de patrones)*

4.2.2. Acciones

Del mismo modo que el uso de medidas ayuda a cuantificar la calidad de nuestros modelos, el uso de ciertas acciones sobre los mismos puede ayudar a mejorar su calidad. Por acción entendemos un conjunto de actividades que permite la consecución de una serie de objetivos. En SLAMMER, es posible definir acciones que implementen rediseños para los modelos de un LVDE dado con el fin de mejorar su calidad o eliminar determinados errores de diseño. También es posible especificar acciones de tipo más general (elaborar un informe, imprimir un modelo, etc.) con el objetivo de incrementar la funcionalidad del entorno generado mediante meta-modelado.

La parte del meta-modelo de SLAMMER que permite la definición de acciones se muestra en la figura 4.45. Una acción (clase *Action*) está formada por una secuencia ordenada de tareas (clase *Task*). Cada tarea puede formar parte de distintas acciones. De este modo se facilita la creación de nuevas acciones y el mantenimiento de las existentes ya que cada tarea se especifica una sola vez, se reutiliza tantas veces como sea necesario, y si cambia sólo hay que modificar una única tarea.

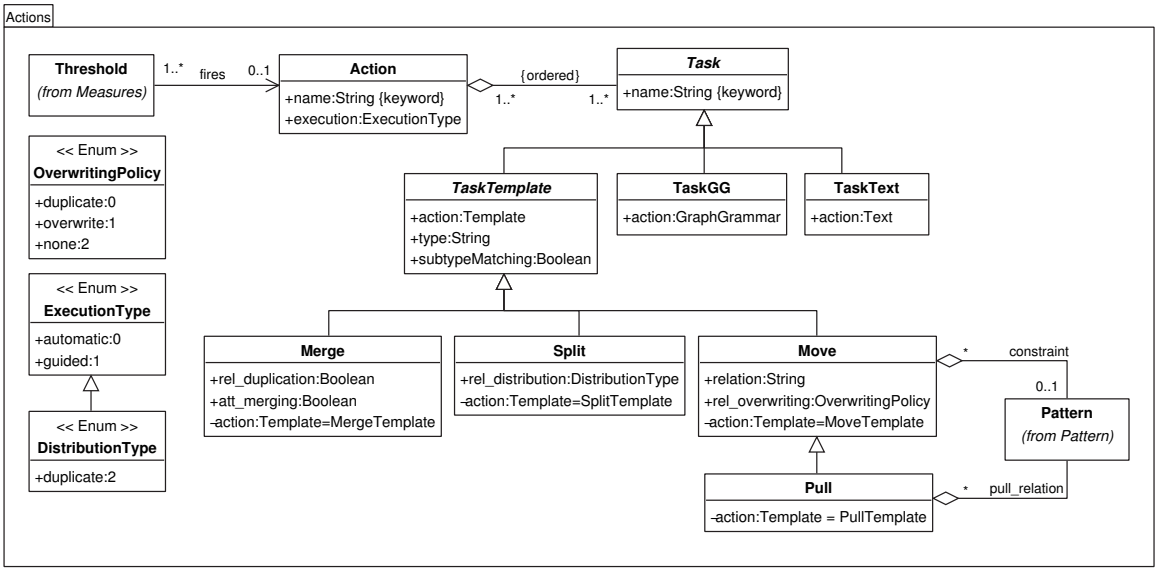


Figura 4.45: *Meta-modelo de SLAMMER (paquete para la definición de acciones)*

Las acciones se llevan a cabo cuando lo decide el usuario, aunque también existe la posibilidad de asociar la ejecución de una acción al indicador de una medida (clase *Threshold* definida en el paquete *Measures* de la figura 4.41). En ese caso, la acción se ejecuta para aquellos valores del dominio de la medida que cumplen la condición asociada al indicador. Sin embargo, ya que en muchos casos la ejecución de rediseños de manera automática puede acarrear problemas [35], las acciones disponen de un atributo *execution* que permite

seleccionar el tipo de ejecución. El atributo puede tomar los siguientes valores:

- *automatic*: en este caso la ejecución se realiza automáticamente para cada valor del dominio. Esto es útil si la acción corrige un error de diseño, y por tanto su aplicación es obligatoria.

- *guided*: en este caso, antes de realizar la acción se pedirá confirmación al usuario para cada valor del dominio donde vaya a aplicarse. Esto resulta útil cuando el indicador es sólo uno de los diversos factores que guían la aplicación de la medida. De este modo, la medida se utilizaría para detectar las oportunidades de rediseño, aunque quizás no todos se lleven a cabo.

Como se puede apreciar en la figura 4.45, SLAMMER define tres mecanismos para la especificación de tareas: código textual (clase *TaskText*), sistemas de transformación de grafos [61, 133] (clase *TaskGG*), y plantillas genéricas predefinidas para la manipulación de modelos (clases concretas que heredan de *TaskTemplate*). Cada uno de estos mecanismos se explica a continuación.

En primer lugar, SLAMMER permite la definición de tareas mediante código, lo cual es útil para construir tareas complejas o que no implican manipulación de modelos, como puede ser la generación de un informe.

El segundo mecanismo para el modelado de tareas son las gramáticas de grafos, ya que permiten expresar de manera formal y visual computaciones en grafos. Se pueden usar como alternativa al código para expresar rediseños de manera gráfica, de tal modo que no se imponga al diseñador de la tarea el aprendizaje de un lenguaje procedimental específico para su modelado ni de librerías gráficas para la manipulación de modelos. Además, el mantenimiento de los rediseños expresados mediante gramáticas de grafos resulta en muchos casos más intuitivo y, por tanto, menos costoso de realizar. Por estas razones hay bastantes propuestas de diversos autores para el rediseño de modelos utilizando lenguajes de transformación de modelos y gramáticas de grafos [99, 104, 133]. En el capítulo 5 se mostrarán algunos ejemplos.

Finalmente, SLAMMER también permite modelar tareas mediante plantillas genéricas predefinidas que pueden adaptarse y ser utilizadas para un LVDE dado. Esas plantillas realizan tareas básicas que son habituales al trabajar con lenguajes visuales: fusionar dos elementos en uno, dividir un elemento en dos, o mover una relación desde un elemento a otro donde, quizás, los elementos origen y destino están relacionados. Los cuatro tipos de manipulación se ilustran en la figura 4.46. Aunque las tareas consideradas son muy básicas, son tan generales que pueden aplicarse a cualquier LVDE. Además, se pueden combinar varias, incluso con otras especificadas mediante gramáticas de grafos o texto, para definir acciones complejas. El objetivo es facilitar la definición de manipulaciones

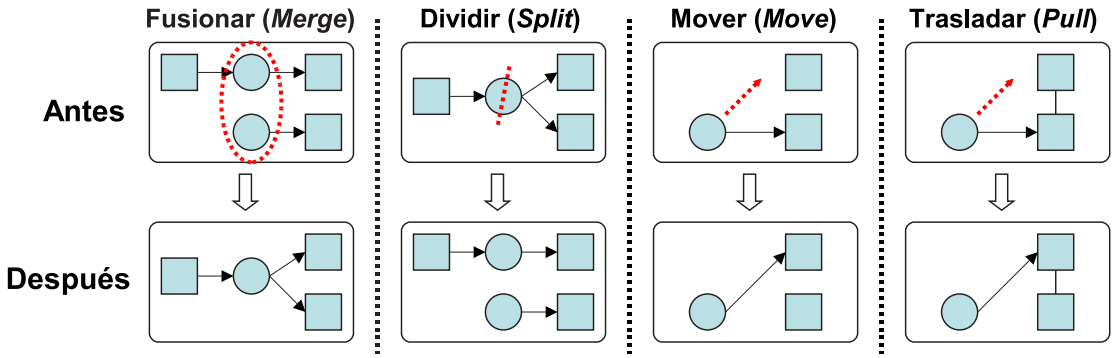


Figura 4.46: *Algunos tipos de manipulación en lenguajes visuales*

de modelos usuales de tal modo que el diseñador de la tarea tenga que realizar el menor trabajo posible.

Los cuatro tipos de manipulación que muestra la figura 4.46 se corresponden con las clases *Merge*, *Split*, *Move* y *Pull* del meta-modelo de SLAMMER, respectivamente. Todas ellas heredan de una clase común denominada *TaskTemplate* que define un atributo de tipo *Template* para definir la plantilla de código que implementa la tarea (atributo *action*), así como el tipo de los elementos a los que se aplica (atributos *type* y *subtypeMatching*, este último para indicar si la tarea también se aplicará a los subtipos de *type*). Cada clase concreta tiene una plantilla predefinida específica, de tal modo que para configurar una de esas tareas para un LVDE dado sólo hay que definir el tipo de elemento al que se aplicará y seleccionar una serie de parámetros de ejecución que son específicos de cada tarea, y que se detallan a continuación. Posteriormente, el capítulo 5 ilustrará su uso con algunos ejemplos.

La tarea *Merge* fusiona dos entidades del mismo tipo (atributo *type*) en una sola que reúne las relaciones de las entidades originales (véase figura 4.46). La tarea se lleva a cabo siempre y cuando el modelo resultante cumpla las restricciones de cardinalidad definidas por su meta-modelo. Sendos atributos permiten configurar la fusión de relaciones y atributos en la entidad resultante. El primero de ellos (*rel_duplication*) establece qué política seguir cuando las entidades iniciales definen una relación al mismo elemento, en cuyo caso se puede optar bien por dejar que la entidad resultante tenga esa relación dos veces (cada una proveniente de una de las entidades originales), o bien dejar sólo una de ellas. El segundo parámetro de ejecución (atributo *att_merging*) especifica la política de fusión de atributos, que puede realizarse bien concatenando su valor en las entidades originales, o bien tomando el valor de uno de ellos. Como puede observarse, la política de ejecución es la misma para todas las relaciones y atributos del tipo de entidad para la que se define. Una alternativa (no contemplada en el meta-modelo) sería asignar a cada relación y atributo su propia política de ejecución. Sin embargo, no se ha optado por esa solución ya que desde el punto de la usabilidad podría ser tedioso de especificar por parte del diseñador de la

tarea. Desde el punto de vista operacional, en la solución por la que se ha optado podrían especificarse excepciones a la política de ejecución general mediante la definición de tareas adicionales que serían ejecutadas antes o después de la fusión.

La tarea *Split* toma una entidad del tipo especificado y la sustituye por dos nuevas del mismo tipo que se reparten las relaciones de la inicial. En cierto modo podemos decir que la tarea "divide" en dos una entidad. El valor de los atributos en las entidades que se crean coincide con el de la entidad inicial, anexando un número secuencial incremental si son clave. En cuanto a la política de distribución de las relaciones, el atributo *rel_distribution* permite definir si las relaciones de la entidad original se replicarán en las dos nuevas entidades, o si por el contrario se distribuirán entre ambas ya sea de manera aleatoria o según criterio del usuario. Estas opciones se corresponden con los valores *duplicate*, *automatic* y *guided* del parámetro, respectivamente.

La tarea *Move* se utiliza para mover relaciones entre dos entidades del mismo tipo (véase figura 4.46). Para configurar esta tarea hace falta definir el tipo de las entidades involucradas (atributo *type*), el tipo de la relación a mover (atributo *relation*) y la política de sobrescritura a seguir en caso de que una relación ya exista en la entidad destino (atributo *rel_overwriting*). Valores posibles para la política de sobrescritura son *duplicate* si se quiere mover las relaciones manteniendo las existentes en el destino, de tal modo que es posible que haya relaciones duplicadas en la entidad destino tras realizar el movimiento; *overwrite* si se quiere sobrescribir aquellas relaciones que ya existan; y *none* si no se quiere mover aquellas relaciones que ya existen en el destino. En la figura 4.47 se muestra la ejecución de una tarea *Move* definida para el tipo de entidad "cuadrado" y el tipo de relación "x", donde se ilustra los distintos resultados obtenidos dependiendo de la política de sobrescritura aplicada.

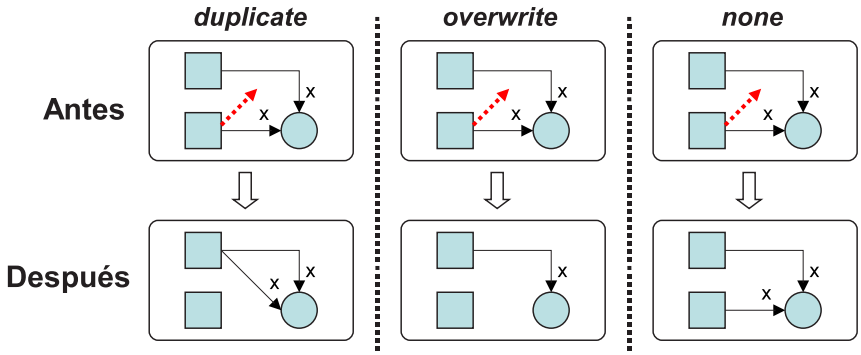


Figura 4.47: *Políticas de sobrescritura para la tarea* Move

En principio, todas las relaciones del tipo especificado se mueven de la entidad origen a la entidad destino. Sin embargo, es posible restringir el número de relaciones a mover mediante la utilización de un patrón que contenga restricciones adicionales que deben

cumplirse en el modelo para poder realizar el movimiento (relación *constraint* en el meta-modelo). Dicho patrón recibe como parámetros los elementos involucrados en la tarea, es decir, la relación que se quiere mover y las relaciones origen y destino. En ese caso sólo se mueven las relaciones para las que existe alguna instancia del patrón.

Por último, la tarea *Pull* es una especialización de la tarea *Move* en la cual se exige que exista cierto tipo de relación entre las entidades origen y destino. Dicha relación se especifica mediante un patrón (relación *pull_relation*) que tiene como parámetros las entidades origen y destino. De ese modo, una relación sólo se moverá si existe la configuración especificada en el patrón.

Integración de mecanismos de consistencia en procesos de rediseño

Como se ha explicado anteriormente, en SLAMMER las acciones se pueden usar para especificar rediseños con distintos objetivos, tales como mejorar alguna propiedad de los modelos, corregir errores de diseño o reestructurar los modelos hacia patrones de diseño conocidos. En el caso de sistemas especificados mediante un conjunto de diagramas, como ocurre al utilizar LVDEs multi-vista, resulta crucial mantener la consistencia entre los distintos modelos después de aplicar un rediseño sobre cualquiera de ellos. Es decir, si uno de los modelos cambia como resultado de la ejecución de una acción especificada en SLAMMER, los cambios deben propagarse al resto de modelos implicados o relacionados con él.

El meta-modelo de SLAMMER no incluye mecanismos explícitos para gestionar la consistencia tras la ejecución de acciones. Tales mecanismos forman parte de la semántica del lenguaje, y por tanto es el proceso de definición del LVDE el que debe permitir establecer dependencias entre los puntos de vista de un lenguaje y proporcionar herramientas para su gestión. Por ello, en vez de incluir mecanismos de consistencia en el meta-modelo de SLAMMER, se ha optado por integrar los definidos en el marco de soporte para lenguajes multi-vista que se presentó en la sección 4.1, de tal modo que no sólo la modificación de modelos por parte del usuario provoque la ejecución de reglas de consistencia y consecuente propagación de cambios, sino también la ejecución de acciones de SLAMMER que tengan como resultado una modificación de alguno de los modelos del sistema.

La integración de ambos marcos implica un estudio sobre qué tipos de vista pueden tener asociado un modelo SLAMMER. En principio todos los tipos (vistas del sistema, semánticas, derivadas y dirigidas a audiencia) son susceptibles de ser medidos y rediseñados. Sin embargo, excluimos las vistas semánticas ya que éstas no forman parte del lenguaje que el usuario maneja y conoce, sino que son herramientas internas de análisis. Por el contrario las medidas y acciones deben llevarse a cabo sobre los modelos escritos con la notación que el usuario del lenguaje utiliza para que puedan serle útiles. En cuanto a las vistas derivadas y dirigidas a audiencia, aunque pueden usarse para medir o rediseñar una parte del sistema,

sólo permiten la propagación de cambios desde el modelo base a la vista. Es decir, que si un rediseño se realiza sobre una vista orientada a audiencia, los cambios no se propagarán al resto del sistema. Para solventar esto sería necesario ampliar la definición de vista derivada y orientada a audiencia con un conjunto de reglas adicionales que permitiera la propagación de cambios desde la vista al modelo base. Desde allí, el resto de reglas de consistencia se encargaría de la propagación al resto de modelos. Finalmente, cualquier vista del sistema puede tener asociado un modelo SLAMMER, y los cambios producidos en las vistas como resultado de un rediseño se propagarían mediante las reglas de consistencia definidas en el marco multi-vista. Un caso particular de vista del sistema es el repositorio, que resulta útil para poder definir medidas y acciones que necesitan información dispersa en distintos modelos del sistema (pero unificada en el repositorio). El problema es que las reglas de consistencia definidas en el marco multi-vista son insuficientes si una acción ejecutada sobre el repositorio añade nuevos elementos al mismo (aunque bastan para tratar la modificación y eliminación de elementos). Esto es debido a que el marco considera que la creación de elementos por parte del usuario siempre se realiza en las vistas del sistema pero no en el repositorio, y por tanto no existen reglas que especifiquen a qué vista del sistema incorporar los elementos que se crean en este último. Aunque esto no llegaría a dejar el sistema en un estado inconsistente, el problema es que las nuevas clases y asociaciones creadas tras el rediseño podrían pasar desapercibidas al usuario si no aparecen en ninguna de las vistas del sistema. Por tanto, la solución integradora requiere incrementar el conjunto de reglas de consistencia para considerar este nuevo caso, así como un algoritmo para la creación de nuevas vistas que incluyan los nuevos elementos si no es posible añadirlos a ninguna de las existentes.

La figura 4.48 muestra la estructura de las reglas adicionales que se necesitan para propagar a alguna vista del sistema los elementos creados en el repositorio por acción de un rediseño. En concreto, por cada punto de vista del LVDE se generan la primera y última reglas para cada clase de su meta-modelo, y la segunda regla de la fila superior para cada asociación. A modo de ejemplo, la figura muestra las reglas para una clase genérica de nombre “Class” y una asociación llamada “assoc”. Estas reglas se añaden a las existentes en la relación de consistencia para la propagación de cambios que va desde el repositorio al punto de vista.

La primera regla de la figura 4.48 propaga las nuevas clases creadas en el repositorio (es decir, aquellas cuyo atributo *refcount* es 0) a una vista del sistema que, en principio, puede ser cualquiera que permita ese tipo de clase. Como la regla modifica el valor del atributo *refcount*, ésta sólo se aplica una vez y por tanto la clase se añade a una sola vista en el sistema. La regla de su derecha hace lo mismo pero para incluir las nuevas asociaciones en alguna vista. Nótese que, cuando se ejecutan estas reglas, la única posibilidad de que el atributo *refcount* de un elemento del repositorio valga 0 es que el elemento haya sido creado por un rediseño. Si la clase apareciese en otra vista, *refcount* sería distinto de cero.

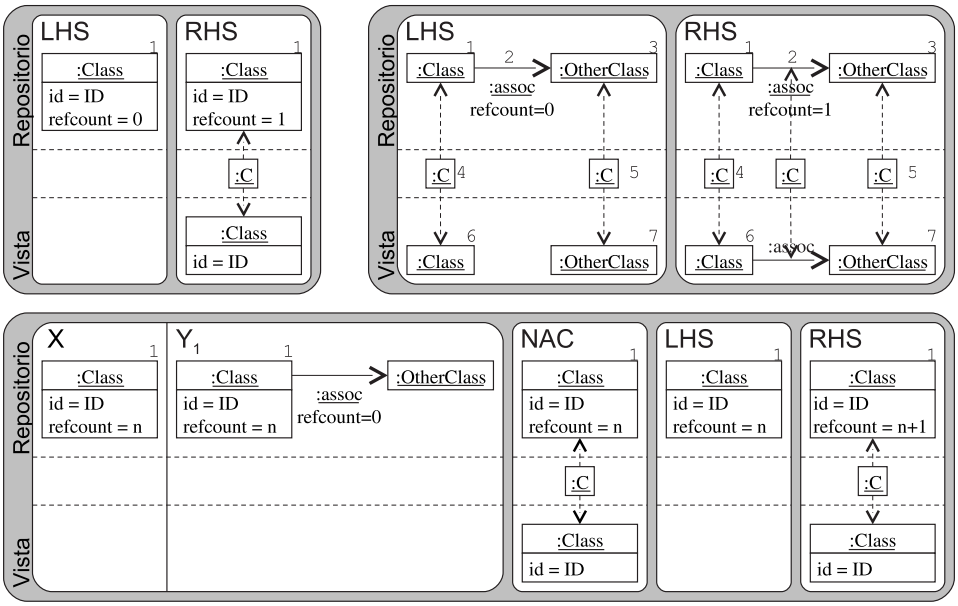


Figura 4.48: *Reglas de consistencia adicionales para rediseños en el repositorio*

Por otro lado, tampoco es posible que sea 0 debido a que la clase se haya eliminado de una vista y el TGTS que construye el repositorio le haya asignado ese valor, ya que entonces ese mismo TGTS se habría encargado de borrar el elemento del repositorio antes de realizar la propagación de cambios.

Por otro lado, un rediseño también puede crear clases en el repositorio que tengan asociaciones de entrada o salida nuevas. Si la clase se propaga a una vista que no admite el tipo de la nueva asociación, ésta última no se llegaría a propagar nunca. Para evitar que esto ocurra se utiliza la última regla de la misma figura, que propaga una clase a una vista si no existe en la vista (NAC), y si además tiene alguna asociación de entrada o salida nueva de alguno de los tipos que admite el tipo de vista (condición de aplicación X-Y_1). La condición de aplicación de esta regla depende de cada clase, ya que tendría un grafo consecuencia Y_i por cada tipo posible de asociación. De ese modo, aunque la clase se haya propagado previamente a otra vista, también se propagará si es necesario porque queden asociaciones nuevas sin propagar.

Un segundo problema asociado a la propagación es qué hacer cuando hay clases o asociaciones nuevas para las que aún no se ha creado una vista que admita esos tipos, de tal modo que las reglas anteriores no pueden llegar a aplicarse. En esos casos se necesita crear una nueva vista del sistema del tipo apropiado donde añadir estos elementos. De este modo, el algoritmo para recuperar la consistencia del sistema tras un rediseño sobre el repositorio constará de los siguientes pasos:

1. Aplicar las reglas de consistencia del repositorio a cada una de las vistas del sistema

(incluidas las reglas que muestra la figura 4.48).

2. Obtener el conjunto e de elementos del repositorio que no aparecen en ninguna vista (es decir, aquellos cuyo atributo *refcount* es 0). Sea e_t el conjunto de tipos de los elementos de e.

3. Mientras haya elementos en e:

   a) Obtener el punto de vista vp cuyo meta-modelo incluye el mayor número de tipos de e_t.

   b) Crear una nueva vista v conforme al punto de vista vp.

   c) Aplicar las reglas de consistencia del repositorio a la vista v.

   d) Recalcular los conjuntos e y e_t.

4.2.3. Sintaxis concreta

Para representar un modelo SLAMMER se ha optado por una sintaxis concreta visual en vez de una textual. La razón es que los conceptos definidos en SLAMMER se pueden representar fácilmente mediante un diagrama que muestre gráficamente el conjunto de medidas y acciones definidas para un LVDE, así como su relación a través de indicadores.

En la sintaxis concreta diseñada para SLAMMER, las medidas se representan mediante rectángulos blancos con el tipo y nombre de la medida dentro. Las dependencias entre medidas se visualizan mediante flechas, donde la cabeza de la flecha indica la dirección del flujo de datos. Los indicadores se representan como triángulos amarillos con un signo de exclamación dentro y el nombre debajo, y están relacionados mediante segmentos con las medidas que los definen. Si el indicador dispara una acción también estará unido a ella mediante un segmento. Las acciones son círculos verdes con una flecha dentro y el nombre de la acción debajo. Si el modo de ejecución es automático, un segundo círculo rodea la figura. Por último las tareas se representan como elipses con el tipo y nombre de la tarea en su interior. Las tareas que corresponden a una acción están unidas a ella mediante segmentos numerados.

La figura 4.49 muestra un ejemplo de modelo SLAMMER utilizando su sintaxis concreta, donde puede verse cómo la utilización de una representación visual resulta más intuitiva que una textual para obtener una visión general del modelo SLAMMER. El modelo contiene la definición de cuatro medidas, donde una de ellas (`Permission Inh Factor`) utiliza el resultado de otras dos (`Subject Permissions` y `Subject Inh Perms`). La medida `Depth_of_Node` define un indicador que dispara automáticamente una acción formada por una sola tarea especificada mediante una gramática de grafos.

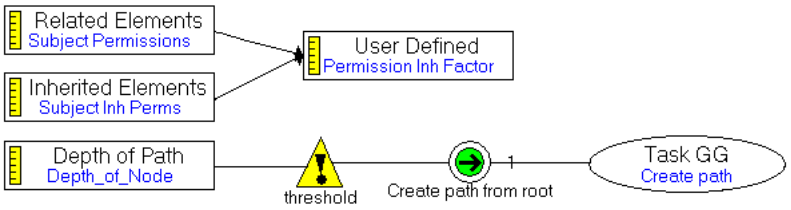


Figura 4.49: *Ejemplo de modelo SLAMMER utilizando la sintaxis concreta*

4.2.4. Implementación de la propuesta

Como se dijo al comienzo de esta sección, el objetivo de SLAMMER es proporcionar a los desarrolladores de entornos visuales una herramienta que facilite la integración de mecanismos para medir y mejorar la calidad de los modelos especificados en tales entornos. Por esta razón se ha construido una herramienta que permite especificar modelos SLAMMER asociados a un LVDE y generar automáticamente a partir de ellos herramientas de medición y rediseño que se integran en el entorno visual.

La herramienta se ha construido usando el entorno de meta-modelado AToM^3 [51]. En AToM^3, la sintaxis abstracta del LVDE se especifica mediante un meta-modelo, mientras que la sintaxis concreta se especifica asociando un icono a cada clase del meta-modelo y un tipo de flecha a cada asociación. En el caso de SLAMMER, su sintaxis abstracta está formada por la suma de los meta-modelos mostrados en la figuras 4.41, 4.42, 4.44 y 4.45. Este meta-modelo se ha enriquecido con atributos orientados a la configuración del entorno generado desde un modelo SLAMMER. En concreto, tal como muestra la figura 4.50, se ha añadido la clase abstracta *UIButton* de la que heredan las clases *Measure*, *Action* y *Task*. Esta clase tiene como único atributo el booleano *button* el cual, si toma el valor cierto, significa que se añadirá un botón para ejecutar la medida, acción o tarea en el entorno generado. Esto es útil, por ejemplo, para prevenir la ejecución directa de medidas auxiliares que sólo se usan para facilitar el cálculo de otras. También se han añadido dos atributos a la clase *Measure* que permiten configurar cómo se realizará la generación de informes. El primero de ellos, *genReport*, permite seleccionar si se generará un informe con los resultados obtenidos en una medición. Por defecto esto es así, pero puede haber casos en que la medida se utilice para detectar oportunidades de rediseño y disparar la ejecución de acciones, de tal modo que no se necesite obtener un informe intermedio. El segundo atributo, *report*, se utiliza para determinar si los informes a generar contendrán el resultado de la medición para todos los valores del dominio, o sólo los de aquellos que cumplen los criterios establecidos por algún indicador.

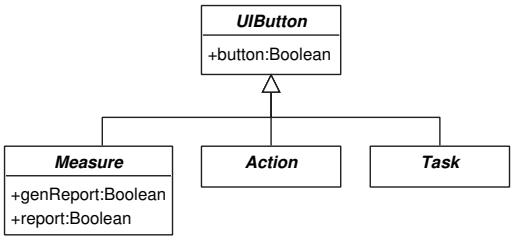


Figura 4.50: *Meta-modelo de SLAMMER (propiedades para la configuración del entorno)*

A los elementos del meta-modelo de SLAMMER se les asignó la sintaxis concreta mostrada en el apartado 4.2.3. A partir de esta definición, AToM^3 generó automáticamente un

entorno visual para la edición de modelos SLAMMER. A este entorno se le incorporó un generador de código capaz de sintetizar herramientas de medición y rediseño a partir de los modelos. Finalmente, el entorno se integró en la interfaz de AToM^3 para permitir la definición de medidas y rediseños para los LVDEs especificados mediante meta-modelado en AToM^3. La figura 4.51 muestra el entorno generado para SLAMMER, donde se están configurando los atributos de una medida de tipo *DepthOfNode* para cierto LVDE. La edición de los atributos se realiza en el cuadro de diálogo numerado como "2". Si entre ellos hay alguno que es de tipo patrón, la herramienta abre una nueva ventana para poder especificar el patrón adecuado dependiendo del dominio, como muestra la ventana "3" para el caso del atributo *step*.

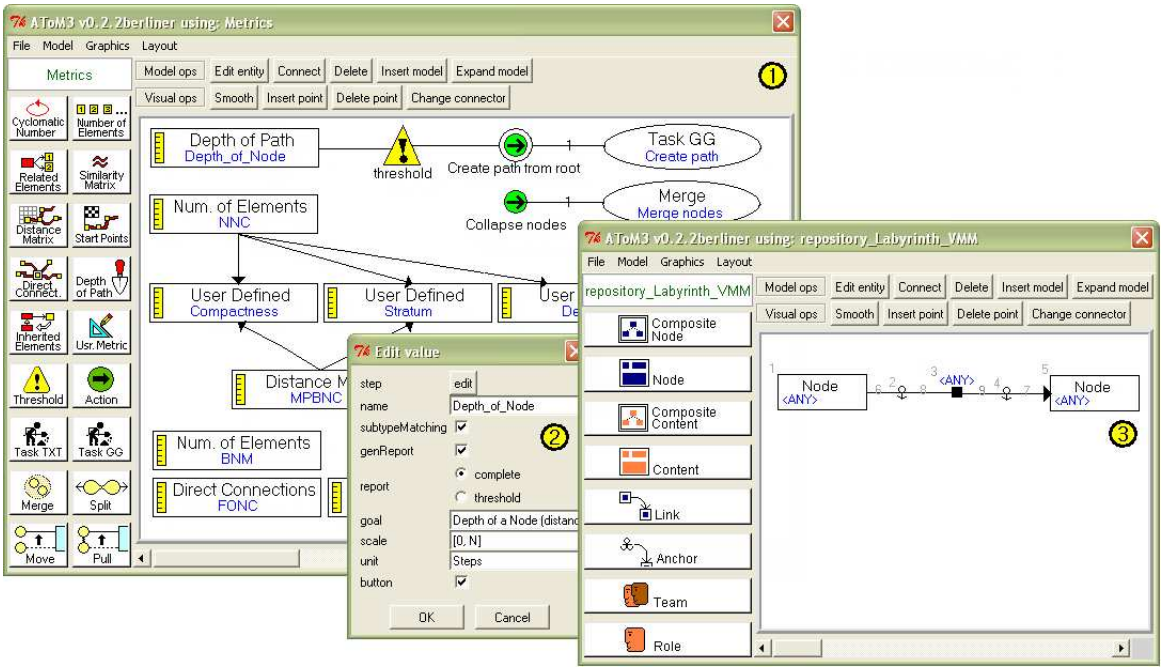


Figura 4.51: *Herramienta de soporte para SLAMMER*

A modo de esquema, la figura 4.52 muestra los pasos para la definición, generación y uso de un entorno visual para un LVDE utilizando AToM^3 y la herramienta de soporte para SLAMMER que integra. En la parte izquierda se muestra el proceso de definición del entorno visual, para lo que el diseñador del LVDE debe especificar la sintaxis abstracta del lenguaje mediante un meta-modelo, y su sintaxis concreta mediante la asignación de una apariencia visual a cada elemento del meta-modelo (paso 1). En el caso de lenguajes multi-vista, el diseñador también debe especificar los puntos de vista del lenguaje como subconjuntos del meta-modelo global. La intersección de cada pareja de puntos de vista establece la clase de dependencias que puede surgir entre los modelos conforme a esos tipos (véase la sección 4.1 para más información). Además, un experto en calidad puede espe-

cificar un modelo SLAMMER con las medidas y acciones aplicables al lenguaje particular (pasos 2a y 2b). Las medidas se definen configurando los elementos de tipo *Measure* definidos en SLAMMER, para lo que hay especificar su dominio (tipos) y cómo los atributos de interés para cada medida se expresan en el lenguaje particular (patrones). Las acciones están compuestas de tareas que se especifican mediante gramáticas de grafos, dando valor a los atributos de un conjunto de plantillas predefinidas, o mediante código (Python). Además, se pueden asociar a indicadores de medidas de tal modo que la acción se realice sólo para aquellos valores del dominio cuya medición cumpla la condición del indicador. En el caso de lenguajes multi-vista, lo habitual es que el modelo SLAMMER se asocie al repositorio ya que las mediciones y rediseños suelen requerir información del sistema completo y no de las vistas aisladas (aunque este último caso también puede darse). Finalmente, aunque en la figura se muestran los roles de diseñador del LVDE y experto en calidad, en muchas ocasiones es la misma persona quien desempeña ambas actividades.

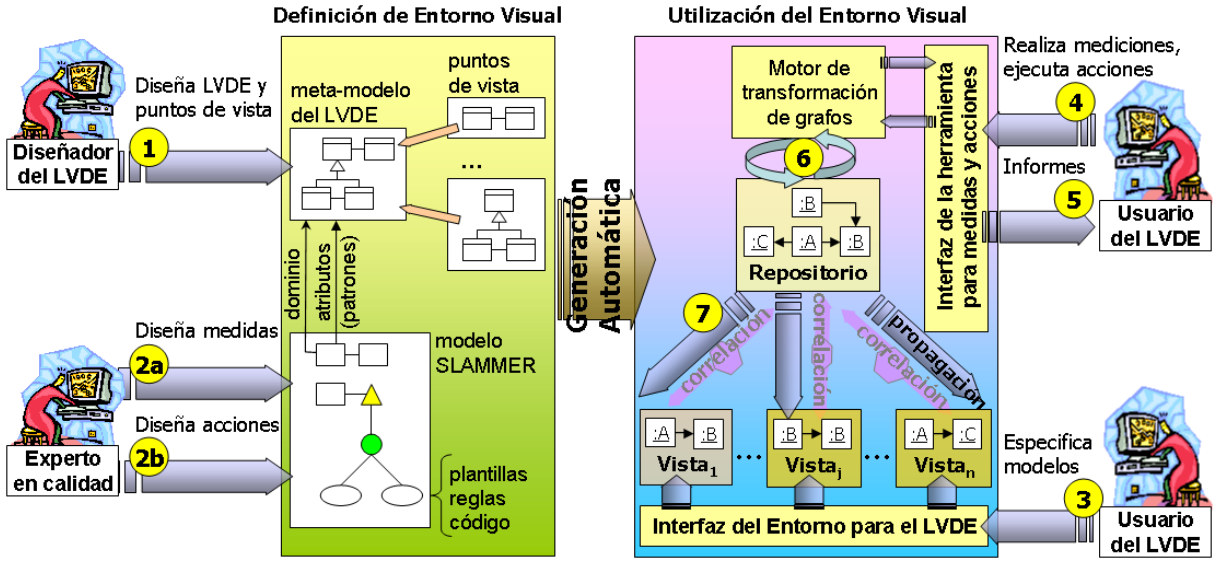


Figura 4.52: *Esquema del enfoque propuesto*

A partir de esa descripción de alto nivel, AToM^3 genera automáticamente un entorno visual para el LVDE cuyas funcionalidades se muestran en la parte derecha de la figura 4.52. El entorno generado lo utiliza el "usuario del LVDE". El rol particular ostentado por tal usuario dentro de un proyecto software dependerá de la funcionalidad del LVDE en cada caso particular. Por ejemplo, en algunos casos el usuario será un diseñador de software si el lenguaje está orientado al modelado de sistemas, o un desarrollador si el lenguaje se va a utilizar para generar código siguiendo un enfoque dirigido por modelos. También es posible que haya distintos roles de usuario usando simultáneamente el entorno generado, por ejemplo si el objetivo del lenguaje es establecer un vocabulario común entre los participantes de un proyecto para favorecer la discusión y el entendimiento del dominio de aplicación.

Independientemente de quién es el usuario del LVDE en cada caso, éste interactúa con la interfaz del entorno visual generado para editar modelos (paso 3). El entorno comprueba que los modelos son conformes al meta-modelo del lenguaje o al de sus puntos de vista en el caso de lenguajes multi-vista. En este último caso, el entorno garantiza la consistencia sintáctica y semántica entre los distintos diagramas mediante el uso de TGTSs que construyen un repositorio interno que fusiona los distintos modelos a través de sus elementos comunes. En el proceso de fusión se crean relaciones entre los elementos del repositorio y los de los modelos de los que son copia, de tal forma que los cambios realizados en un modelo pueden propagarse al repositorio, y desde allí a otros modelos si es necesario (véase la sección 4.1 para más información). El entorno visual incorpora una interfaz generada a partir del modelo SLAMMER previamente especificado que permite realizar mediciones de los modelos y ejecutar acciones sobre los mismos (paso 4). Esta interfaz está integrada en el entorno visual, y disponible en el repositorio en el caso de lenguajes multi-vista. A partir de los resultados de las mediciones se generan informes que se almacenan como ficheros con extensión pdf (si la medida se configuró con generación de informes, paso 5). Las acciones se ejecutan sobre el repositorio y, si éstas lo modifican, los cambios se propagan desde el mismo a los modelos del sistema utilizando las reglas de consistencia que se usan para la propagación de cambios realizados por el usuario (pasos 6 y 7).

Como ejemplo, la figura 4.53 muestra el entorno visual generado para un LVDE multi-vista. La ventana del fondo permite la edición de vistas del sistema, y la ventana superior corresponde al repositorio. Los botones de color amarillo se generaron automáticamente desde un modelo SLAMMER definido para dicho lenguaje, y permiten la medición y ejecución de acciones sobre el repositorio. En el capítulo 5 se puede consultar una descripción más detallada de este ejemplo.

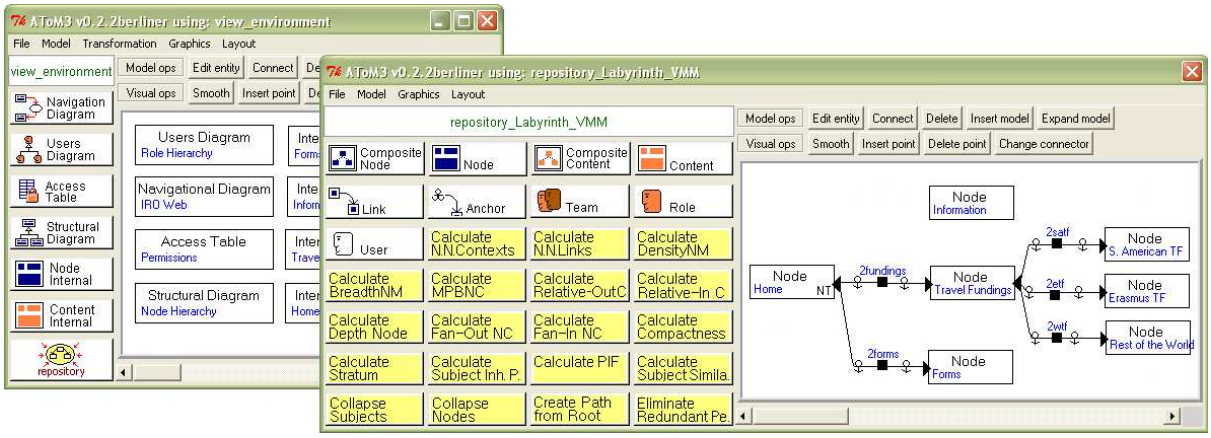


Figura 4.53: *Entorno visual generado (los botones amarillos corresponden a mecanismos de medición y rediseño generados desde un modelo SLAMMER)*

En la figura 4.54 se muestran algunos informes generados desde el entorno visual con

resultados de mediciones. Dichos resultados se muestran en forma de tabla donde cada registro corresponde a un valor en el dominio. El primer informe corresponde a una medida orientada a modelo, y por tanto el informe contiene un único registro con el resultado de la medición. En cambio el segundo informe (que se muestra parcialmente) corresponde a una medida orientada a grupo, y por tanto cada registro contiene un valor del dominio y el resultado de la medición para el mismo.

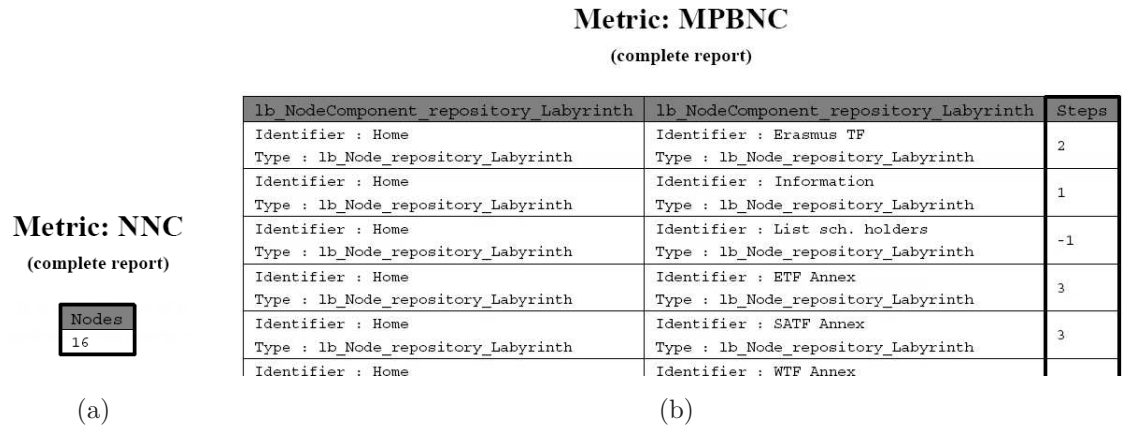


Figura 4.54: *Ejemplo de informes generados*

4.3. Conclusiones

En este capítulo se ha presentado un marco formal para facilitar la definición e integración de mecanismos de control de la calidad en entornos visuales. El marco se basa en técnicas visuales y formales, como son el meta-modelado o la transformación de grafos.

La primera parte del capítulo se dedicó a los LVDEs multi-vista. Para definirlos se propuso utilizar un meta-modelo del que los puntos de vista del lenguaje son subconjuntos. A partir de esta definición se generan automáticamente TGTSs que garantizan la consistencia sintáctica de las vistas: cuando una vista cambia, los cambios se copian a un repositorio, desde donde se propagan al resto de vistas. Además es posible generar distintas reglas para obtener diversos paradigmas de comportamiento. Para analizar la consistencia de la semántica dinámica de las vistas o verificar propiedades se propusieron las vistas semánticas. Éstas son una parte del sistema expresada en algún formalismo que proporciona técnicas de análisis. El diseñador del LVDE debe definir un TGTSs que transforme la vista a analizar al formalismo seleccionado, el mecanismo de análisis para verificar la propiedad, y un patrón triple para anotar el resultado obtenido en la vista semántica de nuevo a la notación original. También se presentaron los patrones visuales de consulta como lenguaje declarativo de consulta sobre modelos. A partir de los patrones se genera un TGTS que construye el modelo resultante de la consulta, y mantiene sincronizados el modelo consultado y el modelo resultante si se producen cambios en el primero.

La segunda parte del capítulo presentó un LVDE denominado SLAMMER que incluye generalizaciones de algunas métricas de producto y manipulaciones de modelos frecuentes. Las métricas se pueden configurar para un LVDE concreto mediante patrones que identifican los elementos a medir. Para especificar acciones (o rediseños) se pueden personalizar plantillas genéricas predefinidas, o bien usar transformación de grafos. Además, SLAMMER permite definir nuevas métricas y acciones diferentes de las propuestas, componer métricas para el cálculo de otras más complejas, y ejecutar acciones en base al valor de ciertas métricas.

El capítulo también presentó un prototipo que implementa el marco propuesto sobre la herramienta de meta-modelado AToM^3. El prototipo permite complementar el meta-modelo de un LVDE con puntos de vista, vistas semánticas, vistas orientadas a audiencia, métricas y rediseños. A partir de esta definición se genera un entorno visual que integra los mecanismos definidos.

Capítulo 5

Evaluación

En el capítulo previo se definió un marco para la integración de herramientas orientadas a medir y mejorar la calidad de modelos específicos de dominio en entornos visuales que incluía el soporte de LVDEs multi-vista mediante transformación de grafos triples, su verificación mediante transformaciones a dominios semánticos y posterior anotación de resultados, y la definición de medidas y acciones mediante el uso de patrones visuales. En este capítulo se aborda la validación empírica y analítica de dicho marco mediante su aplicación práctica en la definición y desarrollo de entornos visuales para LVDEs en las áreas del modelado de sistemas web e hipermedia, y en el de la especificación de bibliotecas digitales. También se presenta el entorno generado para un pequeño subconjunto de UML, con el objetivo de estudiar la factibilidad de la solución presentada en su aplicación a lenguajes visuales de propósito general. En base a los resultados obtenidos, el siguiente capítulo evaluará la consecución de los objetivos planteados al comienzo de este documento, así como la generalidad y riqueza expresiva de la solución.

El capítulo consta de tres secciones. Las dos primeras corresponden a la evaluación empírica de las soluciones presentadas en las secciones 4.1 y 4.2 respectivamente. En la primera sección se presentan algunos entornos visuales multi-vista generados mediante la herramienta de soporte integrada en AToM^3 que se presentó en el apartado 4.1.5. Se presentarán en detalle los entornos generados para los LVDEs multi-vista Labyrinth [53] y VisMODLE [126], así como para un pequeño subconjunto del lenguaje de propósito general UML [182]. También se incluye una breve descripción de otros entornos generados siguiendo el mismo enfoque.

En la segunda sección se muestra la aplicación de SLAMMER en el modelado de un conjunto de medidas y rediseños para Labyrinth. Aunque definidas sobre el mismo LVDE, se presentarán medidas y rediseños orientados a dos campos distintos (dentro del diseño web) como son el modelado de la navegación y las políticas de seguridad. Igualmente, se presentará el entorno generado a partir de esta definición utilizando para ello la herramienta de soporte integrada en AToM^3 que se presentó en el apartado 4.2.4.

Para terminar, la última sección realiza una evaluación analítica de la herramienta desarrollada como soporte tecnológico del marco, mediante su comparación con otras herramientas de meta-modelado.

5.1. Evaluación empírica del soporte para lenguajes visuales de dominio específico multi-vista

El objetivo de esta sección es evaluar empíricamente el marco presentado en la sección 4.1 mediante la construcción de entornos visuales para diversos LVDEs multi-vista. Para ello se muestra el proceso de creación de dichos entornos en AToM^3 a partir de la especificación de los puntos de vista, vistas semánticas y vistas dirigidas a audiencia de cada lenguaje. Los entornos generados integran mecanismos automáticos de consistencia sintáctica entre las vistas del sistema, herramientas de análisis y verificación cuyos resultados se muestran en términos del lenguaje visual, y permiten obtener vistas parciales del sistema predefinidas en el sistema (orientadas a audiencia) o como resultado a una consulta sobre el mismo.

Los dos primeros apartados de esta sección muestran la definición y generación de sendos entornos visuales para dos LVDEs multi-vista en distintos dominios de aplicación. El primer lenguaje se denomina Labyrinth [53] y permite el diseño de aplicaciones web. El segundo lenguaje se denomina VisMODLE [126] y se utiliza para el diseño de bibliotecas digitales. En este segundo caso, el entorno generado integra un simulador y un generador de código que permite la síntesis automática de código a partir de los modelos de diseño. A continuación, el tercer apartado intenta mostrar la generalidad del enfoque mediante la generación de un entorno visual para un pequeño subconjunto del lenguaje de propósito general UML. Para finalizar, un último apartado recoge brevemente otros entornos visuales construidos en AToM^3 siguiendo el mismo enfoque.

5.1.1. Labyrinth

Labyrinth [53] es un LVDE para el diseño de aplicaciones web e hipermedia. La figura 5.1 muestra una parte de su meta-modelo con la definición de las características más relevantes. Brevemente, en Labyrinth una aplicación web se modela mediante un conjunto de nodos (clase abstracta *NodeComponent*) donde se colocan contenidos de información tales como texto, imágenes, etc. (clase abstracta *ContentComponent*). Nodos y contenidos admiten relaciones de composición para permitir la definición de estructuras de información complejas. Las propiedades de navegación se especifican mediante enlaces (clase *Link*) y anclas (*Anchor*) asociadas a los nodos y contenidos que actúan como origen y destino del enlace, con las siguientes restricciones: los contenidos no pueden ser destino de un enlace, y sólo pueden ser origen de los que definen los nodos que los contienen. El modelado de usuarios se basa en el paradigma de control de acceso basado en roles (RBAC), e incluye usuarios (clase *User*), roles (clase *Role*) y equipos (clase *Team*). Los roles representan una función de trabajo dentro de una organización, y los equipos representan una colección heterogénea de roles y equipos con un interés común. Roles y equipos disponen de una serie de permisos sobre los objetos del sistema (relación *PA*), y se pueden agrupar para formar estructuras jerárquicas donde los permisos asignados a los roles más generales los heredan los roles más específicos, y los permisos asignados a los equipos se propagan a todos sus componentes. De este modo no hace falta definir explícitamente todos los permisos del sistema, sino que éstos se propagan a través de la jerarquía. Los usuarios asumen roles de donde obtienen el conjunto de permisos que indican a qué nodos y contenidos se les permite acceder. Aunque no se les puede asignar equipos, participan en ellos a través de la jerarquía de herencia definida para sus roles.

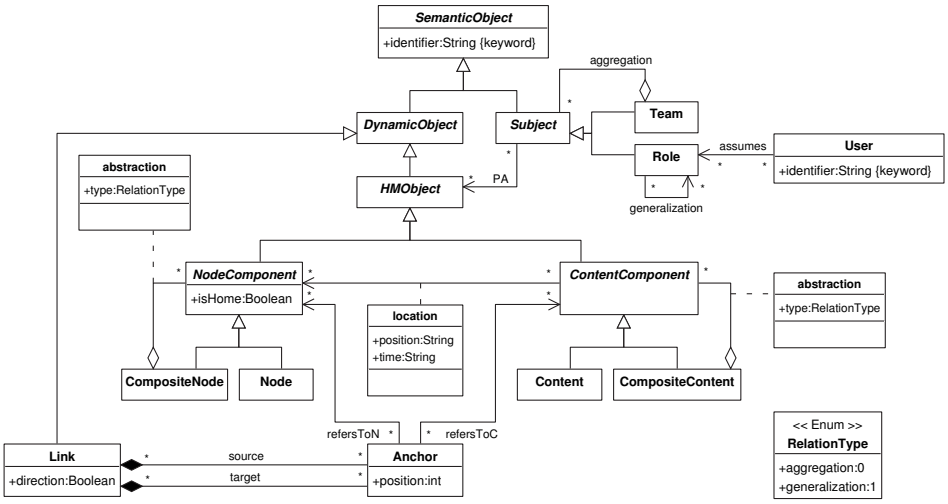
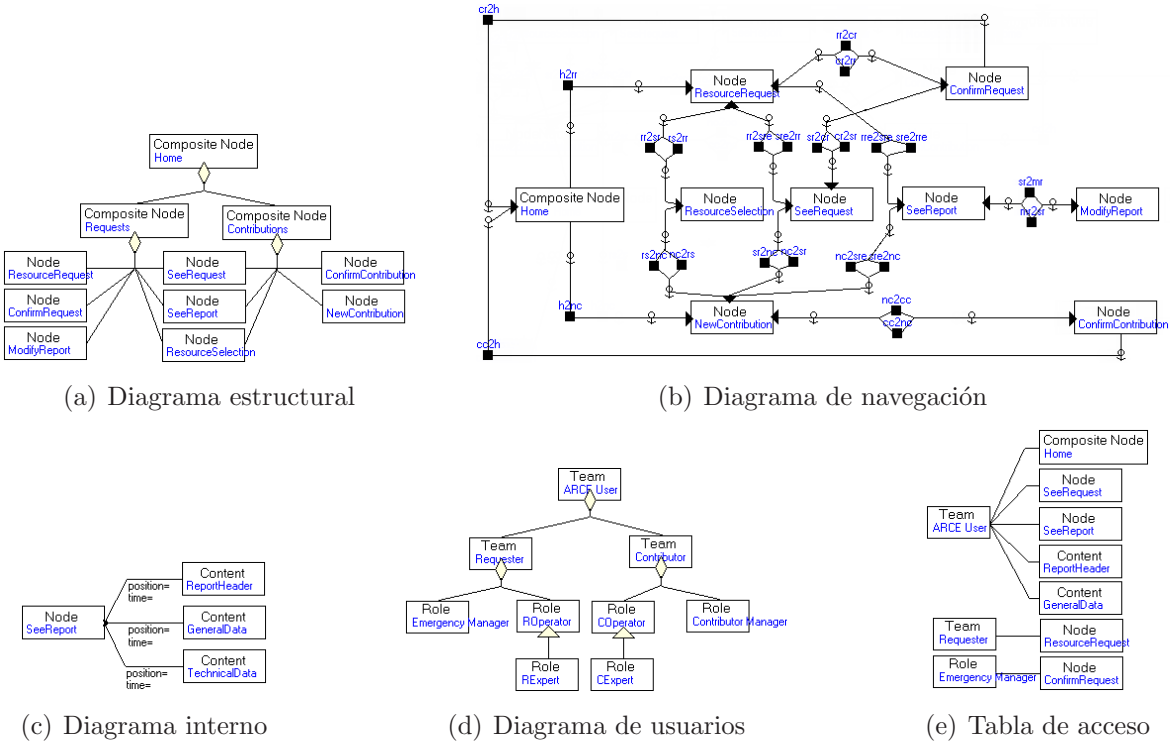


Figura 5.1: *Meta-modelo de Labyrinth*

El Método de Desarrollo Ariadne (ADM) [54] utiliza los conceptos del meta-modelo de Labyrinth para definir un conjunto de puntos de vista o tipos de diagrama que capturan los distintos aspectos involucrados en un diseño web. El método consta de tres fases: *diseño conceptual*, donde se identifican los tipos abstractos de componentes, relaciones y funciones de la aplicación; *diseño detallado*, donde se detalla su comportamiento, procesos y características; y *evaluación*, donde se evalúa la usabilidad del sistema diseñado mediante la generación y estudio de prototipos. Este apartado se centra en el diseño conceptual, el cual propone seis puntos de vista para modelar distintos aspectos abstractos de una aplicación web. La figura 5.2 recoge un ejemplo de cada uno de ellos. Los diagramas pertenecen al diseño de una parte del sistema de cooperación internacional ARCE [10] para la gestión de recursos en casos de desastre.



(a) Diagrama estructural (b) Diagrama de navegación

(c) Diagrama interno (d) Diagrama de usuarios (e) Tabla de acceso

Figura 5.2: *Diagramas de diseño conceptual en el Método de Desarrollo Ariadne*

El *diagrama estructural* de una aplicación (figura 5.2(a)) contiene la definición de sus nodos y relaciones estructurales. En la figura, los nodos hoja corresponden a páginas web que serán accesibles en la aplicación final, mientras que los nodos compuestos agrupan nodos que realizan el mismo tipo de actividad. El *diagrama de navegación* (b) captura los caminos de navegación expresados como enlaces entre los nodos. La flecha del enlace indica el sentido de la navegación. Además, cada nodo y contenido puede tener un *diagrama interno de nodo o contenido*, respectivamente, que describe su estructura interna. Por ejem-

plo, la figura 5.2(c) muestra el diagrama interno del nodo `SeeReport`, el cual está formado por tres contenidos. El *diagrama de usuarios* (d) recoge la jerarquía de roles y equipos definidos en la aplicación. Finalmente, la *tabla de acceso* (e, mostrada parcialmente) especifica a qué nodos y contenidos pueden acceder los usuarios que ostentan cierto rol o pertenecen a determinado equipo. Tener acceso a un nodo no implica tenerlo automáticamente sobre sus contenidos, sino que se debe indicar de manera explícita en la tabla de acceso. Por ejemplo, el rol `ARCE User` tiene permiso sobre el nodo `SeeReport` y sus contenidos `ReportHeader` y `GeneralData`, pero no lo tiene para el contenido `TechnicalData` (que también pertenece al nodo) porque la tabla de acceso no lo especifica. Por otro lado, la tabla sólo recoge las asignaciones directas de permisos; implícitamente, cada rol y equipo también hereda los permisos asignados a sus ancestros.

Como puede verse, ADM establece una familia de LVDEs definidos en base al meta-modelo de Labyrinth. Cada lenguaje individual está definido según su propio meta-modelo, el cual es un subconjunto de ese meta-modelo. Así pues, se ha usado AToM^3 para construir un entorno visual que soporte el diseño conceptual de ADM y proporcione mecanismos de consistencia y análisis de las vistas de un diseño. Dicho entorno y su proceso de definición se muestran a continuación.

Definición del entorno para Labyrinth

Para construir el entorno primero se definió el meta-modelo completo de Labyrinth en AToM^3, y a continuación se usó la herramienta para la especificación de lenguajes multi-vista que éste integra (véase apartado 4.1.5 para una explicación detallada sobre las prestaciones de la herramienta). En la herramienta se definieron los seis puntos de vista del lenguaje, así como una vista semántica para el análisis de propiedades en el formalismo redes de Petri P/T. La figura 5.3 muestra esta definición. El punto de vista central llamado `repository_Labyrinth` y las relaciones de consistencia entre el mismo y los demás puntos de vista los generó automáticamente la herramienta.

La vista semántica se incluyó con el objetivo de verificar la corrección de la política de acceso de un diseño Labyrinth en términos de la disponibilidad de nodos, contenidos y caminos de navegación para diversos roles [89]. Para definir la vista semántica no se necesitó especificar el meta-modelo del formalismo redes de Petri, sino sólo su nombre (es decir, ya se disponía de una implementación previa del formalismo construida en AToM^3). Además se definió un TGTS que construye, a partir del repositorio, una red de Petri que simula el comportamiento de un sujeto dado, ya sea éste un rol o un equipo. El TGTS crea un lugar por cada nodo y contenido para el que el sujeto tiene permiso de acceso, y transforma los enlaces de navegación entre ellos en transiciones de la red. El sujeto se representa con una marca en la red; de este modo, el que una marca esté en un lugar significa que el sujeto está accediendo al nodo o contenido que el lugar representa. Visitar

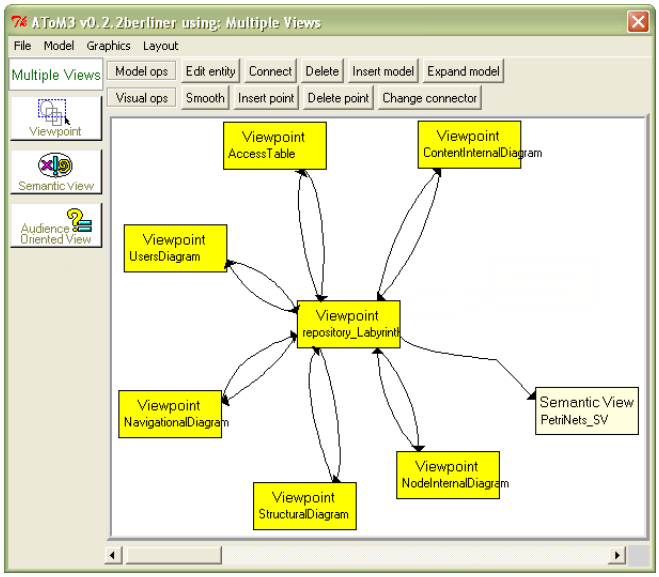


Figura 5.3: *Definición de los puntos de vista y vistas semánticas de Labyrinth*

un nodo implica visitar todos sus contenidos, y por tanto el marcado de la red de Petri equivale al conjunto de nodos y contenidos accedidos en cierto momento por el sujeto.

Las reglas del TGTS que construyen la vista semántica a partir del repositorio se muestran en las figuras 5.4, 5.5 y 5.6 usando una notación compacta. Las reglas se muestran en sintaxis abstracta para permitir al lector identificar fácilmente el tipo de los elementos; sin embargo, la implementación real de las reglas en AToM^3 utiliza la sintaxis concreta de los elementos porque ésta resulta más intuitiva para el diseñador del entorno visual. Las tres reglas de la figura 5.4 crean punteros al sujeto bajo estudio y a cada uno de sus ancestros (realizan un aplanado de la jerarquía de sujetos). En particular, la primera regla crea un nodo de correspondencia con un morfismo al sujeto para el que se calcula la red, cuyo nombre es parámetro de la regla. A continuación, las otras dos reglas suben por la jerarquía de sujetos creando nodos de correspondencia adicionales con un morfismo a cada ancestro del sujeto. Obsérvese que el hecho de usar nodos de correspondencia como elementos auxiliares en la transformación evita tener que modificar el meta-modelo de Labyrinth para incluirlos en el mismo.

La regla *ObjetoALugar* de la figura 5.5 crea un lugar por cada objeto hipermedia (nodo o contenido) para el que el sujeto o sus ancestros (identificados por la ejecución de las reglas previas) tienen permiso de acceso. Esta regla es abstracta, y equivale a las ocho reglas concretas que se obtienen de sustituir el objeto hipermedia y el sujeto por sus subtipos concretos. De este modo constituye una notación más compacta de un conjunto de reglas similares. A su derecha, la regla *EnlaceATransicion* crea una transición de red de Petri por cada enlace de navegación entre objetos permitidos para el sujeto o sus ancestros (esto es,

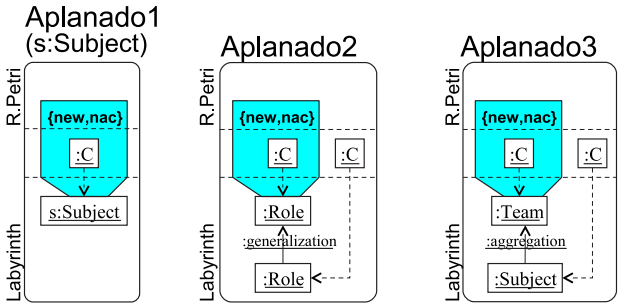


Figura 5.4: *Reglas triples para la transformación de Labyrinth a redes de Petri (i)*

entre objetos que tienen un elemento de correspondencia creado por una ejecución anterior de la regla *ObjetoALugar*). Existe otra regla similar a ésta para el caso en que el origen del enlace sea un contenido en vez de un nodo. La regla *AnclaAArco* crea un arco desde un lugar a una transición si el nodo al que representa el lugar es origen del enlace correspondiente a la transición. Otras dos reglas similares se encargan de crear un arco cuando el origen es un contenido, o cuando el nodo es destino del enlace (creando en ese caso un arco de salida que va desde la transición al lugar).

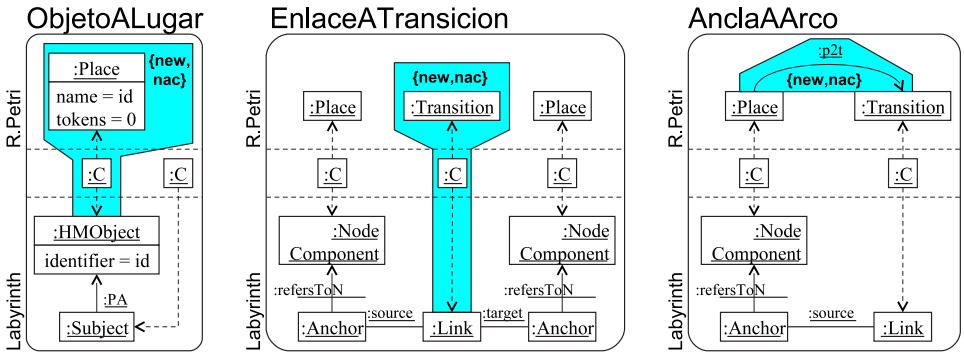


Figura 5.5: *Reglas triples para la transformación de Labyrinth a redes de Petri (ii)*

Cada vez que una transición dispara (esto es, cada vez que se atraviesa un enlace), hay que poner una marca no sólo en el nodo destino del enlace sino también en los contenidos del mismo permitidos para el sujeto. Para ello la regla *UbicacionAArco* de la figura 5.6 crea los arcos a los contenidos apropiados. Por la misma razón, dejar un nodo implica dejar sus contenidos, para lo que otras dos reglas similares crean los arcos necesarios. Finalmente, la regla *SujetoAMarca* crea un lugar adicional con una marca que representa al sujeto, y una transición al lugar que representa la página de inicio de la aplicación. Esta transición modela la entrada del sujeto al sistema. El hecho de representar el estado de inicio como un lugar adicional permite que, cuando el usuario entre al sistema, su marca vaya tanto al lugar del nodo de inicio como a los de sus contenidos, gracias a que la regla anterior

habrá creado los arcos adecuados para ello.

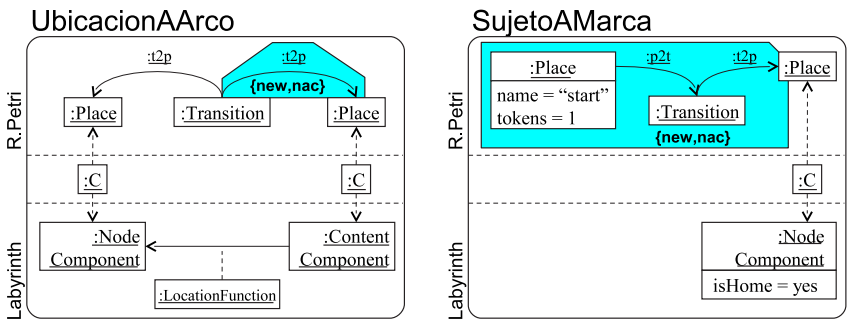


Figura 5.6: *Reglas triples para la transformación de Labyrinth a redes de Petri (iii)*

En conclusión, la aplicación del TGTS crea una red de Petri cuyos lugares corresponden a los nodos y contenidos permitidos para el sujeto que es parámetro de la regla *Aplanado1*, e incluye las transiciones entre ellos. Esto es, la red no contiene lugares para aquellos nodos o contenidos no permitidos al sujeto. Como ejemplo, la figura 5.7 muestra la red de Petri resultante de aplicar el TGTS al rol `Emergency Manager` del sistema ARCE (véase figura 5.2 para consultar las vistas de dicho sistema).

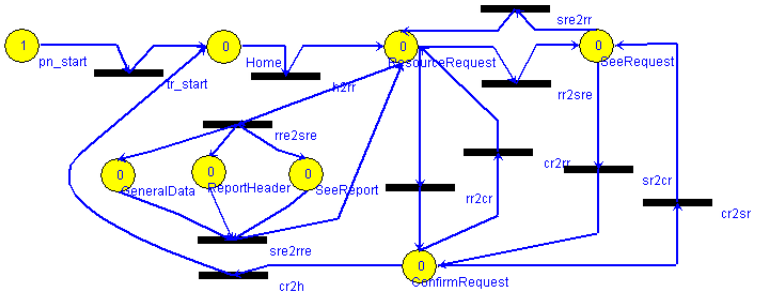


Figura 5.7: *Red de Petri que captura el comportamiento del rol* `Emergency Manager`

Junto al TGTS presentado se definieron seis métodos de análisis para verificar las políticas de seguridad expresadas con Labyrinth, ya que permiten comprobar la existencia o disponibilidad de caminos de navegación, nodos y contenidos en función del rol ostentado por un usuario, verificar que no existen nodos o contenidos inaccesibles para todos los roles y equipos del sistema, así como averiguar si existen nodos terminales sin enlaces de salida para un rol dado.

Estos métodos de análisis usan funciones de análisis (definidas en un API) que calculan el grafo de cubrimiento de la red de Petri resultante de la transformación, y posteriormente usan un *model checker* implementado sobre AToM^3 [47] para verificar la propiedad que corresponde en cada ocasión. Las propiedades se expresan mediante fórmulas CTL que actúan como parámetros de las funciones del API. Las proposiciones atómicas de esas

fórmulas se corresponden con lugares de la red, y son ciertas para un estado de la red si el lugar contiene al menos una marca en dicho estado (véase sección 2.4 para una descripción de la sintaxis de CTL).

El resultado de verificar una propiedad sobre el grafo de cubrimiento da como resultado todos los marcados de la red que satisfacen la propiedad. Las funciones incluidas en el API son: *evalExpression_states*, que obtiene el conjunto de estados de la red que satisfacen la expresión CTL que recibe como parámetro; y *evalExpression_path*, que devuelve las secuencias de transiciones que llevan a un estado que satisface cierta expresión CTL. A continuación se da una descripción de los seis métodos de análisis definidos para Labyrinth:

1. *Un sujeto puede completar cierto camino de navegación.* Esta propiedad permite verificar si un sujeto puede completar una tarea que implica atravesar cierto camino de navegación. La fase de pre-procesamiento del método de análisis obtiene el nombre del sujeto s y la secuencia de pasos de navegación $Np =< nodo_1, nodo_2, ..., nodo_N >$ cuya existencia se quiere verificar. La función de análisis realiza una llamada a la función del API *evalExpression_states* pasando como parámetros la fórmula *E (True U (nodo*$_1$*∧ EX (nodo*$_2$*∧ EX (...))))* que se construye a partir de la secuencia de pasos de navegación a verificar, así como la red de Petri resultante de la transformación para el sujeto s (recuérdese que la red captura el comportamiento de un único sujeto según la política de seguridad establecida). Finalmente, la fase de post-procesamiento muestra un cuadro de diálogo con el mensaje *true* si la función de análisis devuelve un resultado no vacío, y *false* en caso contrario.

2. *Un sujeto puede acceder a cierto nodo o contenido.* Este método de análisis permite detectar qué objetos hipermedia deberían ser accesibles para un sujeto pero no lo son debido a un error de diseño en la política de acceso. También permite verificar que un sujeto no tiene más permisos de los requeridos. Su fase de pre-procesamiento obtiene el sujeto s y el nodo o contenido *hmo* al que se refiere la propiedad. La función de análisis consiste en una llamada a la función *evalExpression_path* pasando como parámetros la fórmula CTL *E (True U hmo)* y la red de Petri para el sujeto s. La función devuelve las secuencias de transiciones que llevan desde el estado inicial de la red al objeto *hmo*.

   Las transiciones obtenidas como resultado del análisis son entrada del patrón triple de anotación que muestra la figura 5.8, y cuya salida es el enlace asociado a la transición más sus nodos origen y destino y anclas correspondientes. De este modo se obtiene sobre el modelo Labyrinth el camino de navegación que lleva al nodo o, si se preguntó por la accesibilidad de un contenido, el camino que lleva a un nodo que lo contiene. Cada secuencia de transiciones resultante se anota separadamente al modelo original (es decir, las soluciones no se muestran todas simultáneamente,

sino que cada solución se muestra por separado). Al igual que se ha hecho con las reglas del TGTS presentado, el patrón utiliza una sintaxis abstracta para facilitar al lector la identificación de tipos; sin embargo, la implementación real de un patrón en AToM^3 utiliza la sintaxis concreta de los elementos ya que resulta más intuitiva para el diseñador del entorno.

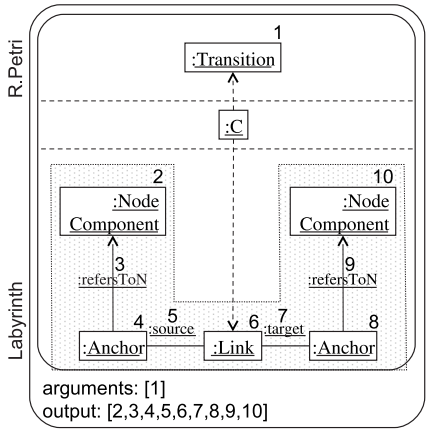


Figura 5.8: *Patrón triple de anotación*

3. *Un nodo o contenido no se muestra a ningún sujeto.* Permite detectar diversos errores de diseño, como por ejemplo la existencia de nodos aislados hasta los que no llega ningún camino de navegación, o de contenidos que nunca se muestran a causa de un error en la asignación de permisos. La fase de pre-procesamiento obtiene el nombre *hmo* del nodo o contenido cuya accesibilidad se quiere comprobar. La función de análisis llama a la función *evalExpression_states* con la fórmula ¬ *E (True U hmo)* para cada sujeto definido en el sistema. La fase de post-procesamiento muestra un cuadro de diálogo con el mensaje *true* si todas las llamadas a la función incluyen el estado inicial como resultado, o la lista de sujetos que pueden acceder al mismo en caso contrario.

4. *Un nodo o contenido se muestra en cada posible camino de navegación para un sujeto.* Permite comprobar que cierto nodo o contenido se le mostrará siempre a determinado sujeto, independientemente del camino de navegación que efectúe. La fase de pre-procesamiento del método obtiene el sujeto *s* y el nodo o contenido *hmo*. La función de análisis llama a la función *evalExpression_states* con la expresión CTL *A (True U hmo)* y la red de Petri obtenida para el sujeto *s*. Si la función incluye en su resultado al estado inicial, entonces es cierto que el nodo o contenido se muestra en cualquier camino que pueda seguir el sujeto. En caso contrario se obtiene un contraejemplo ejecutando la función *evalExpression_path* para la expresión ¬ *E (True U hmo)*. La

función devuelve los caminos de ejecución que llegan a un estado a partir del cual no se muestra *hmo*. En ese caso, la fase de post-procesamiento anota uno de esos caminos (el contraejemplo) al modelo Labyrinth mediante el patrón de la figura 5.8.

5. *Un sujeto nunca queda bloqueado en un nodo.* O en otras palabras, todos los nodos accesibles para un sujeto contienen algún enlace hacia otro nodo de la aplicación. La fase de pre-procesamiento obtiene el sujeto s para el que se quiere verificar la propiedad. La función de análisis realiza una llamada a la función *evalExpression_path* pasando como parámetros la red de Petri correspondiente al sujeto y la expresión CTL *E (True U deadlock)*, donde *deadlock* es un predicado especial que se cumple en aquellos estados que no tienen sucesor. En la fase de post-procesamiento, las secuencias de transiciones obtenidas (es decir, los caminos que llevan a un nodo bloqueado) se anotan al modelo Labyrinth original con el patrón de la figura 5.8.

6. *Un sujeto accede al menos a un contenido de un nodo dado.* Esta propiedad permite detectar aquellos nodos que están vacíos debido a un mal diseño de la política de acceso. La fase de pre-procesamiento obtiene el sujeto s y el nodo n para los que se quiere verifica la propiedad. La función de análisis llama a *evalExpression_states* con la fórmula n y la red correspondiente al sujeto. De este modo se obtienen los marcados de la red que satisfacen la expresión, lo cual incluye tanto el nodo n como todos sus contenidos para los que el sujeto tiene permiso de acceso. Finalmente, los contenidos que se obtienen como resultado de la función de análisis se anotan al modelo original mediante el patrón triple que muestra la figura 5.9.

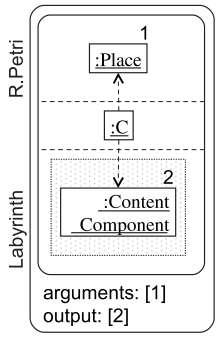


Figura 5.9: *Patrón triple de anotación*

Entorno visual generado para Labyrinth

A partir de la definición de Labyrinth, AToM^3 generó el entorno multi-vista que muestra la figura 5.10. El entorno permite crear vistas de cada uno de los puntos de vista de acuerdo a la multiplicidad especificada, y proporciona mecanismos internos de consistencia

sintáctica entre las mismas. En concreto, la figura muestra la edición del diagrama interno del nodo `SeeReport`.

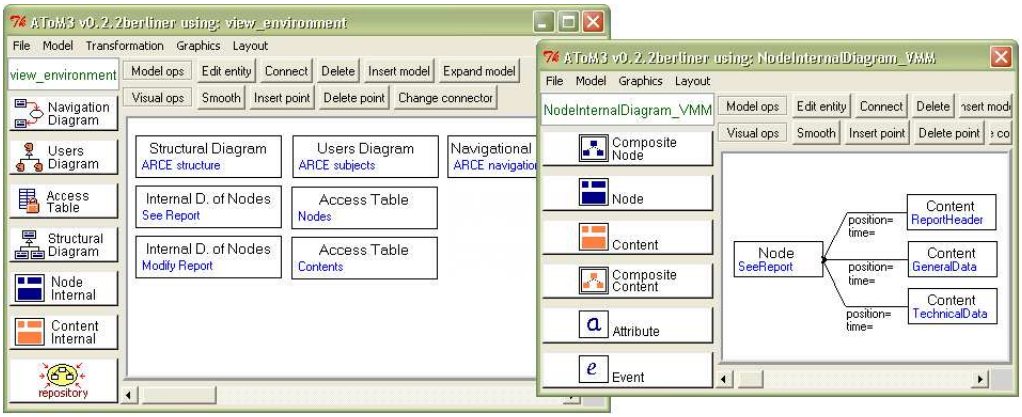


Figura 5.10: *Entorno multi-vista generado para Labyrinth*

El último botón del entorno permite acceder al repositorio del sistema. La interfaz del repositorio incluye botones para crear elementos de Labyrinth, y además seis botones adicionales generados automáticamente a partir de los métodos de análisis definidos. Para verificar una propiedad basta con pulsar el botón correspondiente. Por ejemplo, la figura 5.11 muestra el resultado de un análisis sobre el sistema ARCE (véase figura 5.2 para consultar las vistas de dicho sistema). Los botones de color amarillo corresponden a los métodos de análisis. En concreto la figura muestra el resultado de verificar a qué contenidos del nodo `SeeReport` puede acceder el sujeto `Emergency Manager`, sexta propiedad de las definidas que se ejecuta pulsando el botón `Check Node Contents`. Si se estudian en detalle las vistas del sistema puede comprobarse que, efectivamente, el contenido `TechnicalData` no es accesible para el rol `Emergency Manager` debido a la política de permisos definida, a pesar de que el contenido forma parte del nodo.

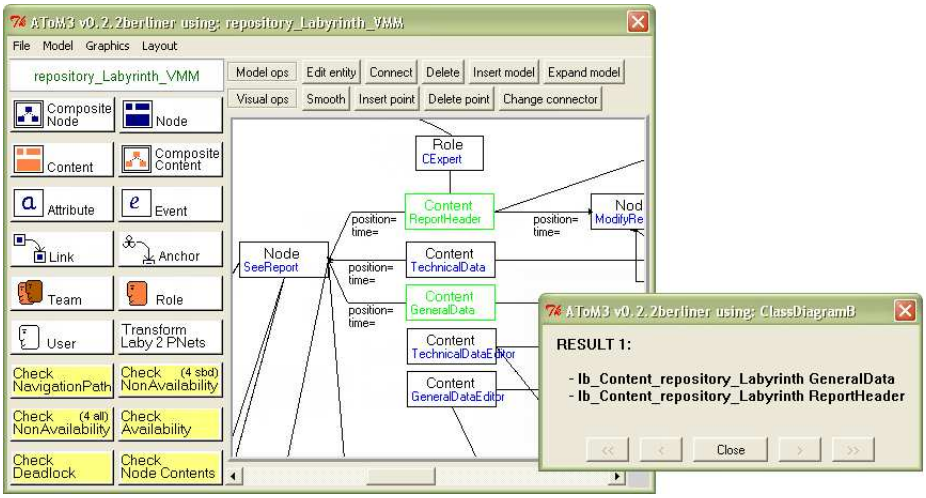


Figura 5.11: *Resultado de verificar una propiedad en el repositorio*

5.1.2. VisMODLE

Este apartado presenta un segundo ejemplo del marco propuesto para la generación de entornos multi-vista, mostrando su aplicación en la construcción de un entorno para el LVDE VisMODLE [126]. Este lenguaje permite el modelado de bibliotecas digitales y, además, los modelos de diseño se utilizan para generar automáticamente parte del código de la biblioteca. La figura 5.12 muestra un esquema de los pasos que se siguieron desde la construcción del entorno para VisMODLE (nivel superior de la figura), pasando por la utilización del entorno para diseñar bibliotecas digitales y validar esos diseños (segundo nivel), hasta la generación de la interfaz de las bibliotecas directamente a partir de los modelos especificados en el entorno (dos últimos niveles). De este modo se comprueba cómo los entornos generados por el marco propuesto se pueden utilizar dentro de un proceso de desarrollo dirigido por modelos, facilitando la especificación y validación de los modelos de un sistema. A continuación se presenta cada uno de los pasos del esquema en detalle.

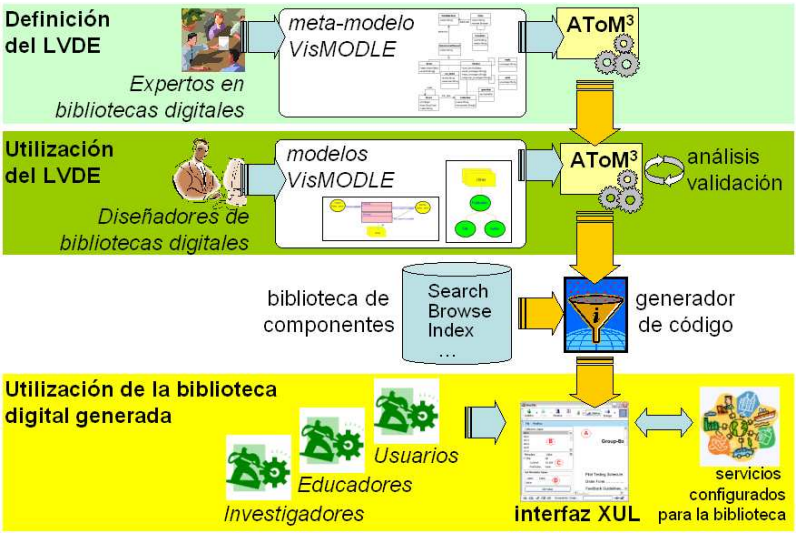


Figura 5.12: *Arquitectura dirigida por modelos para el lenguaje VisMODLE*

En primer lugar, VisMODLE fue diseñado por Alessio Malizia en colaboración con expertos en el dominio de las bibliotecas digitales. El lenguaje contiene cinco tipos de diagrama, cada uno de ellos orientado a la captura de un aspecto distinto del sistema: la información multimedia proporcionada por la biblioteca (*modelo de colecciones*); la estructura de dicha información (*modelo estructural*); los servicios que implementan el comportamiento de la biblioteca (*modelo de servicios*); la interacción entre los actores y servicios del sistema (*modelo social*); y, finalmente, el comportamiento individual de cada actor y servicio expresado como una máquina de estados (*modelo de comportamiento*). La figura 5.13 muestra el meta-modelo completo del lenguaje.

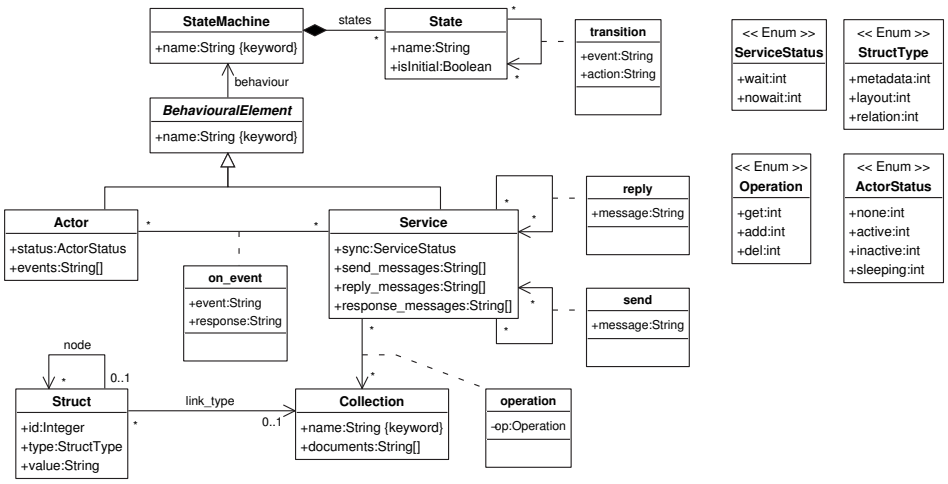


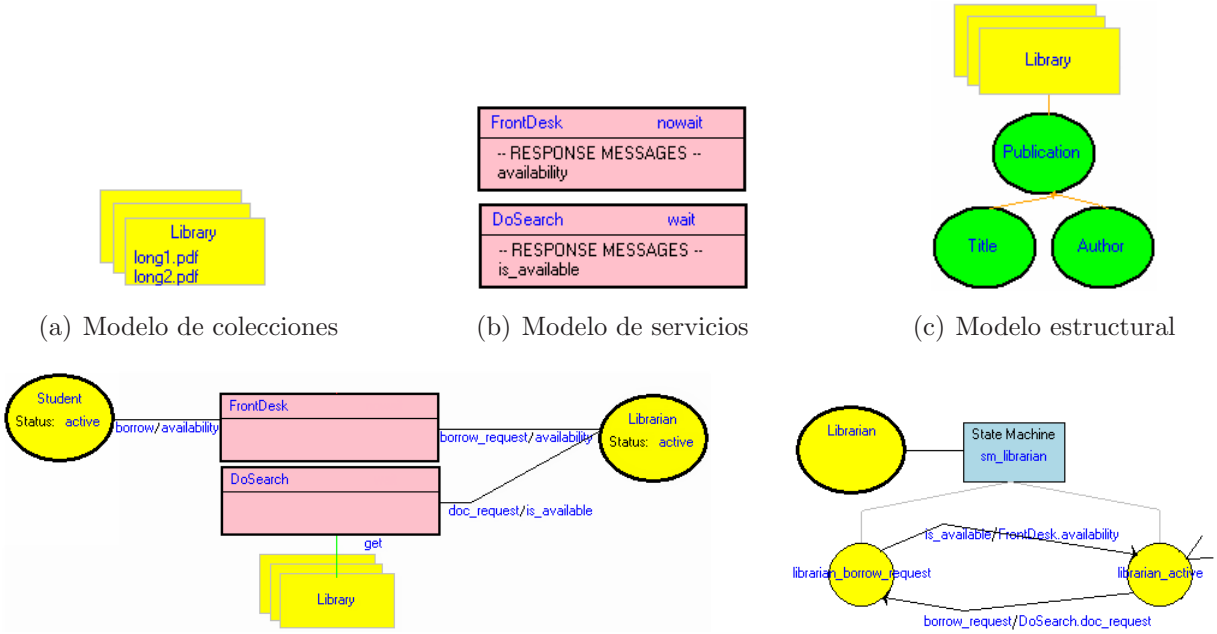
Figura 5.13: *Meta-modelo de VisMODLE*

Para ilustrar el modelado de bibliotecas digitales con VisMODLE, así como los tipos de diagrama que define, la figura 5.14 muestra un conjunto de modelos que corresponden al diseño de una pequeña biblioteca universitaria. Para empezar, el *modelo de colecciones* especifica los elementos almacenados en la biblioteca, y por tanto la única entidad relevante del meta-modelo para este tipo de diagrama es la clase *Collection*. La figura 5.14(a) recoge el modelo de colecciones del ejemplo, el cual define una biblioteca llamada `Library` con dos documentos "long1.pdf" y "long2.pdf". De manera similar, el *modelo de servicios* define los servicios que la biblioteca utiliza, y por tanto sólo la clase *Service* es relevante para este tipo de diagramas. El modelo (b) de la misma figura muestra los servicios disponibles para el ejemplo: `FrontDesk`, que permite a los usuarios de la biblioteca solicitar libros y que el bibliotecario les atienda, y `Search`, que se encarga de realizar búsquedas sobre la colección de documentos.

El *modelo estructural* define metadatos (clase *Struct*) sobre los elementos de una colección. Los documentos del ejemplo, tal y como muestra la figura 5.14(c), definen tres metadatos: publicación, autor y título.

El *modelo social* describe el comportamiento general de la biblioteca. Este modelo incluye el conjunto de actores (clase *Actor*) que interactúa con los servicios del sistema, así como las colaboraciones entre servicios y los accesos a las colecciones de documentos. Los actores representan tanto usuarios como componentes hardware o software que usan o soportan servicios de la biblioteca digital. El modelo social para el ejemplo se muestra en la figura 5.14(d).

Por último, los *modelos de comportamiento* especifican la actividad individual de servicios y actores mediante máquinas de estados. Las transiciones de la máquina de estados tienen lugar en respuesta a eventos (esto es, en respuesta a la llegada de un mensaje).



(a) Modelo de colecciones (b) Modelo de servicios (c) Modelo estructural

(d) Modelo social (e) Modelo de comportamiento

Figura 5.14: *Modelado de una biblioteca universitaria usando VisMODLE*

Efectuar una transición puede implicar la ejecución de una acción (esto es, el envío de un mensaje a otro actor o servicio). La figura 5.14(e) muestra el comportamiento del actor `Librarian`. El ejemplo incluye tres modelos de comportamiento adicionales, uno para cada servicio y actor del modelo social.

Definición del entorno para VisMODLE

Al igual que en el caso de Labyrinth, se ha construido un entorno multi-vista que permite modelar bibliotecas digitales con VisMODLE, y además integra mecanismos de consistencia y análisis de las vistas de un sistema [88, 126]. En este caso, además, el entorno incluye un generador de código que sintetiza parte de la biblioteca digital final a partir de la información recogida en los modelos de diseño [126]. Para construir este entorno se empezó especificando el meta-modelo completo de VisMODLE en AToM^3, y a continuación se especificaron sus cinco puntos de vista utilizando la herramienta de especificación de lenguajes multi-vista integrada en AToM^3. La definición del lenguaje incluye una vista orientada a audiencia que obtiene los actores y servicios sin un comportamiento definido (esto es, sin una máquina de estados asociada). Su patrón de consulta es el mostrado en la figura 4.37 del capítulo previo. También se definió una vista semántica para el análisis de propiedades en el formalismo redes de Petri, propiedades que se explicarán más adelante.

La figura 5.15 muestra la definición de los distintos puntos de vista, de la vista semántica, y de la vista orientada a audiencia para VisMODLE. En la figura, el punto de vista que corresponde al repositorio y las relaciones de consistencia se generaron automáticamente.

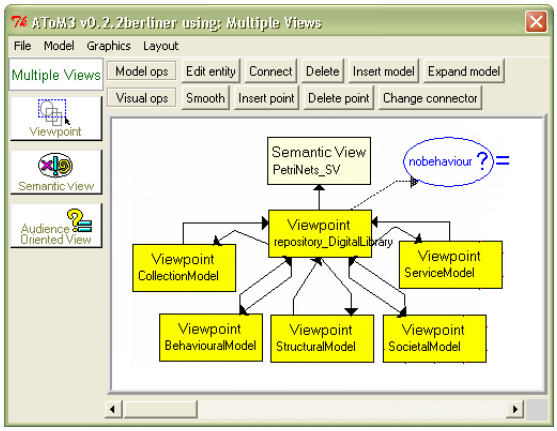


Figura 5.15: *Definición de los puntos de vista, vistas semánticas, y vistas orientadas a audiencia de VisMODLE*

Como se ha comentado anteriormente, los modelos de VisMODLE no constituyen sólo el diseño de la biblioteca digital, sino que también se usan para la síntesis de código. Esto hacía muy importante que su entorno visual fuese capaz de verificar la corrección de los modelos antes de la generación de código. Para ello se definió una vista semántica del repositorio para su análisis en el formalismo redes de Petri P/T, así como el correspondiente TGTS. La transformación traduce cada modelo de comportamiento de un actor o servicio (existente en el repositorio) a lo que se ha denominado "módulo" de red de Petri. La idea es poder simular la ejecución de todas las máquinas de estado "en paralelo". Para ello la semántica de las redes de Petri resulta muy adecuada ya que genera todos los entrelazados posibles.

Las reglas triples de la figura 5.16 pertenecen al TGTS para construir la red de Petri que captura el comportamiento de una biblioteca digital dada. Para ello crean un lugar por cada estado de la máquina de estados (regla *EstadoALugar*), y una transición de red de Petri por cada transición entre estados (regla *TransicionATransicion*). Cuando se crean los lugares, no contienen marcas. Posteriormente, la regla *InicioAMarca* añade una marca en aquellos lugares que corresponden a estados iniciales. En la regla, la tupla de valores asignada al atributo *tokens* del lugar representa el número de marcas antes y después de aplicar la regla, respectivamente. Además de los lugares que corresponden a estados, el TGTS también crea un lugar por cada evento especificado en las transiciones de las máquinas de estados (regla *EventoALugar*). Estos lugares actúan como interfaz del módulo que corresponde al modelo de comportamiento de cada actor o servicio, al que se relacionan por medio de un nodo de correspondencia. La NAC de la regla impide que se cree más de un lugar interfaz para el mismo evento, aunque éste active más de una transición.

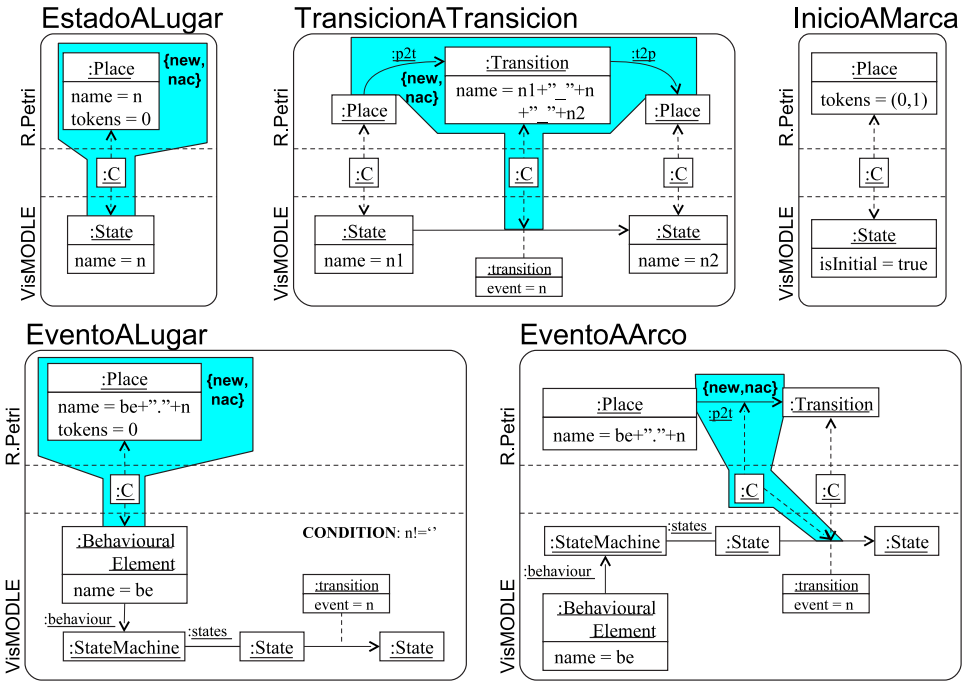


Figura 5.16: *Reglas triples para la transformación de VisMODLE a redes de Petri*

Si una transición de máquina de estados necesita que tenga lugar un evento para su ejecución, entonces la regla *EventoAArco* crea un arco desde el lugar que representa al evento hasta la transición de red de Petri que representa la transición de la máquina de estados. De este modo una transición en la red de Petri final sólo podrá disparar si recibe un evento del tipo apropiado (esto es, si existe una marca en el lugar interfaz apropiado) y además el actor o servicio se encuentra en el estado correcto (esto es, existe una marca en el lugar asociado al estado origen de la transición). De manera similar, el TGTS contiene una última regla que comprueba si una transición define una acción, en cuyo caso crea un arco desde la transición de red de Petri asociada hasta el lugar interfaz del actor o servicio especificado en la acción. De este modo se interconectan los distintos módulos construidos en la red de Petri.

Junto con la vista semántica y el TGTS se definieron seis métodos de análisis para VisMODLE. Éstos permiten comprobar si un servicio o un actor alcanza un estado de bloqueo, ya que por ejemplo lo habitual es que se quiera que los servicios siempre estén disponibles; analizar si la ejecución del sistema termina; detectar si un actor o servicio alcanza cierto estado, y qué estados nunca se alcanzan (lo que sería un error de diseño); obtener qué mensajes se mandan en todos los flujo de ejecución posibles; o analizar si un servicio puede recibir un número ilimitado de mensajes que puedan provocar un desbordamiento. Los métodos de análisis utilizan el mismo API que ya se introdujo en el apartado anterior, el cual incluye las funciones *evalExpression_states* y *evalExpression_path* para obtener

los estados que cumplen una propiedad, o las secuencias de transiciones que llevan a un estado que la cumple, respectivamente. En ambos casos la propiedad se especifica mediante una expresión CTL cuyas proposiciones atómicas son nombres de lugares de la red (véase apartado anterior para una explicación más detallada). A continuación se especifican los seis métodos de análisis definidos para VisMODLE:

1. *Un actor o servicio puede quedar bloqueado.* Este método permite detectar los estados de los que un actor o servicio no puede salir, ya que se queda bloqueado en ellos. Aunque puede ser deliberado, en otras ocasiones puede deberse a que el actor o servicio se queda esperando indefinidamente un mensaje que nunca llega (debido a un error de diseño). La fase de pre-procesamiento del método de análisis obtiene el nombre *be* del actor o servicio. La función de análisis obtiene el conjunto s_i de estados del actor o servicio, y a continuación llama a *evalExpression_states* para cada estado s_i pasando la expresión CTL *A G* (s_i). De este modo se obtienen todas las configuraciones de la red en las que el actor o servicio queda bloqueado (esto es, no se obtiene sólo el estado de bloqueo del actor o servicio, sino también el estado en que se encuentran el resto de elementos del sistema para que el bloqueo ocurra). El marcado de la red se anota al modelo VisMODLE mediante el patrón de la figura 5.17.

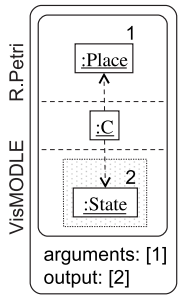


Figura 5.17: *Patrón triple de anotación*

2. *Un actor o servicio alcanza cierto estado.* Este método permite comprobar que un actor o servicio realiza el conjunto de actividades que le lleva a estar en determinado estado. Su fase de pre-procesamiento obtiene el estado s a analizar. La función de análisis consiste en una llamada a *evalExpression_path* para la propiedad s. La función devuelve las secuencias de transiciones que llevan al actor o servicio al estado requerido. Estas transiciones son la entrada del patrón que muestra la figura 5.18, y cuya salida es la transición VisMODLE asociada más sus estados origen y destino.

3. *Un actor o servicio no define estados imposibles.* O en otras palabras, un actor o servicio puede llegar a estar en cualquiera de los estados que define. Este método

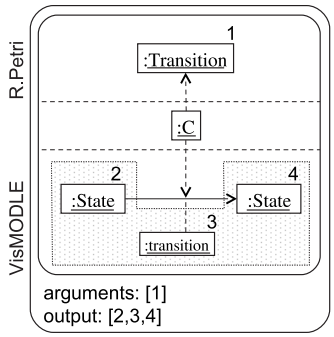


Figura 5.18: *Patrón triple de anotación*

de análisis permite detectar qué estados de un actor o servicio son innecesarios, o bien aquellos que son necesarios pero no llegan a darse debido a un error de diseño. Su fase de pre-procesamiento obtiene los estados s_i definidos para el actor o servicio, que son los que define su máquina de estados. La función de análisis llama a *evalExpression_states* para cada estado, pasando la propiedad s_i correspondiente en cada caso. Finalmente, la fase de post-procesamiento muestra un cuadro de diálogo con el mensaje *true* si todas las llamadas a la función de análisis devuelven resultados. En caso contrario muestra los estados s_i para los que dicha función no devuelve nada, lo que significa que el actor o servicio no puede llegar a estar en este estado.

4. *Un mensaje se envía siempre*. Permite comprobar que una tarea o colaboración siempre se realiza. La fase de pre-procesamiento obtiene el mensaje m y el actor o servicio *be* que lo envía. La función de análisis llama a *evalExpression_states* con la expresión CTL *A (True U be.m)*. Si el estado inicial está en el resultado devuelto por la función, entonces la propiedad se cumple y la fase de post-procesamiento muestra un cuadro de diálogo con el mensaje *true*. En caso contrario se genera un contraejemplo evaluando la función *evalExpression_path* para la expresión $\neg$ *E (True U be.m)*, que calcula las secuencias de transiciones de la red que llevan a un marcado a partir del cual el mensaje no llega a enviarse. En ese caso, la fase de post-procesamiento anota una de esas secuencia al modelo original a modo de contraejemplo, usando para ello el patrón de la figura 5.18.

5. *La biblioteca digital siempre está disponible*. Esto es, la biblioteca no deja de funcionar ni su ejecución queda bloqueada. Para comprobar esto, la función de análisis del método llama a *evalExpression_path* pasando como parámetro la expresión *E (True U deadlock)*. Como se ha explicado anteriormente, *deadlock* es un predicado que se cumple en los estados finales. De este modo la función obtiene las secuencias de transiciones que llevan a un bloqueo de la red, o lo que es lo mismo, a un bloqueo en la ejecución de la biblioteca. Las secuencias de transiciones se anotan al modelo

original mediante el patrón que muestra la figura 5.18.

6. *Un servicio puede sufrir desbordamiento.* Este método de análisis permite analizar si un servicio puede recibir un número ilimitado de peticiones que no tenga tiempo de responder. Su fase de pre-procesamiento obtiene el nombre del servicio *se* a analizar. La función de análisis obtiene el conjunto de mensajes m_i que el servicio puede recibir, y a continuación llama a *evalExpression_states* para cada mensaje pasando como parámetro la expresión CTL $se.m_i[w]$ (el algoritmo usado para calcular el grafo de cubrimiento de una red etiqueta con "[w]" aquellos estados que pueden recibir un número ilimitado de marcas). La fase de post-procesamiento obtiene los mensajes para los que la función de análisis obtuvo algún resultado, y éstos se anotan a VisMODLE usando el patrón de la figura 5.19.

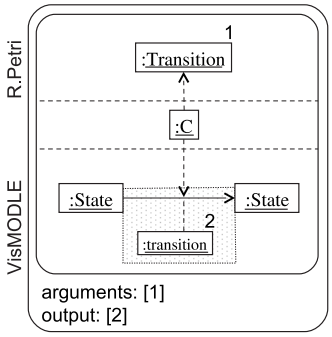


Figura 5.19: *Patrón triple de anotación*

Además de los distintos puntos de vista y métodos de análisis definidos para VisMODLE, el entorno se enriqueció con un simulador [88] que permite animar el repositorio, esto es, visualizar el comportamiento de la biblioteca digital paso a paso. Esto favorece la discusión entre las personas involucradas en la especificación de la biblioteca (que suele ser multidisciplinar, desde bibliotecarios hasta ingenieros de desarrollo), y permite detectar errores de diseño mediante la observación del sistema en "funcionamiento". El simulador anima los modelos de comportamiento de actores y servicios visualizando la recepción y envío de mensajes entre sus máquinas de estados, y comprueba que los eventos y acciones que éstos especifican son coherentes con el intercambio de mensajes especificado en el modelo social. Cabe destacar que la simulación utiliza información de distintas vistas del sistema, y que el hecho de tenerlas unificadas en un repositorio único facilitó la construcción del correspondiente simulador.

Para construir el simulador primero se modificó el meta-modelo de VisMODLE definido en AToM^3 para incluir un conjunto de elementos auxiliares utilizados en la simulación pero no en las vistas del sistema (esto es, los nuevos elementos pueden aparecer en el repositorio

pero en ningún otro tipo de diagrama). Esta es una ventaja de tener un meta-modelo explícito para el repositorio: permite definir elementos auxiliares que no pertenecen al lenguaje y por tanto no intervienen en la definición del lenguaje ni se muestran al usuario final, pero que son útiles para la simulación u otros propósitos. En el caso presente se definieron *mensajes* que los actores y servicios pueden recibir a través de un elemento llamado *entrada* que apunta al estado actual del actor o servicio. La sintaxis concreta de los mensajes es un cuadrado azul con el nombre del mensaje en su interior, mientras que las entradas se representan con un cuadrado negro.

Tras modificar el meta-modelo, se procedió a especificar el simulador mediante una gramática de grafos. El simulador consta de cinco reglas, tres de las cuales se muestran en la figura 5.20. En este caso se ha optado por mostrar las reglas utilizando la sintaxis concreta de VisMODLE para dar una idea más intuitiva del modo en que se animan los modelos ejecutando el simulador. De hecho, en este caso las reglas corresponden a capturas de la especificación real en AToM^3, y por tanto utilizan la notación estándar (LHS-RHS) en vez de la notación compacta.

La regla *CrearEntrada* del simulador crea una entrada para cada actor y servicio definido. Inicialmente la entrada no contiene ningún mensaje y apunta al estado inicial de la máquina de estados del actor o servicio. La regla *ProcesarEvento* realiza un paso de simulación (o transición entre estados) para aquellos casos en que la transición define un evento y ninguna acción. Para que la regla pueda ejecutarse se necesita que el estado origen de la transición sea el estado actual del actor o servicio (es decir, que la entrada apunte a dicho estado), y además en la entrada debe existir un mensaje cuyo nombre coincida con el nombre del evento especificado en la transición. Si esto ocurre, el estado actual del elemento pasa a ser el estado destino de la transición, y el mensaje procesado se elimina de la entrada.

De manera similar, la regla *ProcesarEventoYAccion* de la figura realiza un paso de simulación en los casos en que la transición también define una acción consistente en mandar un mensaje a otro elemento (esto es, la acción tiene la forma "elemento_destino.mensaje"). Esta regla, además de las condiciones previas de la regla anterior, define una condición de aplicación con tres grafos consecuencia y una premisa igual a la LHS de la regla (la figura sólo muestra los grafos consecuencia y los agrupa bajo el nombre PAC). La condición de aplicación comprueba que el elemento destino acepta el tipo de mensaje especificado por la acción. Para poder aplicar la regla se tiene que cumplir una de las tres condiciones de la PAC: o bien el elemento que envía el mensaje es un actor y lo recibe un servicio (primer caso), o bien es un servicio el que manda el mensaje a un actor (segundo caso) o a otro servicio (tercer caso). La regla borra el mensaje de la entrada, cambia el estado actual del actor o servicio, y crea el mensaje especificado por la acción en la entrada del elemento destino. Por último, existen dos reglas adicionales parecidas que consideran transiciones que definen una acción pero ningún evento, y transiciones sin evento ni acción.

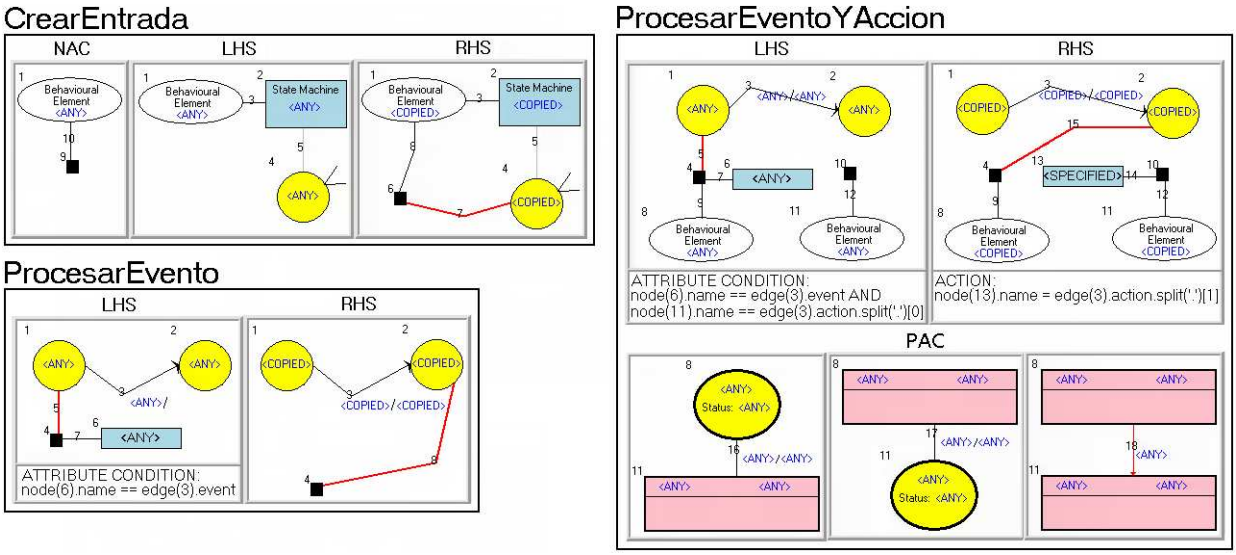


Figura 5.20: *Reglas del simulador construido para el repositorio*

Por último, la definición del entorno para VisMODLE se completó con un generador de código (realizado por A. Malizia) que sintetiza la interfaz de usuario de la biblioteca digital diseñada a partir de los modelos estructural y de colecciones. Ya que AToM^3 permite asociar la ejecución de acciones en respuesta a eventos, la generación de código se asoció al evento de edición de las entidades en dichos modelos. De ese modo, los cambios realizados en los modelos se reflejan automáticamente en la aplicación final. También se definió un proceso por lotes para la síntesis de ese mismo código a partir de modelos existentes. En ambos casos la generación de código consiste en la especialización de un conjunto de plantillas XUL [193] predefinidas según los atributos y relaciones diseñados en la fase de modelado. Estas plantillas dividen la interfaz de usuario de la aplicación final en dos columnas: la primera incluye un navegador para los documentos especificados en la biblioteca digital y permite manipular sus metadatos, mientras que la segunda columna se utiliza para visualizar los documentos de la primera columna. Por otro lado, la información incluida en el modelo social se utiliza para enlazar los servicios que estas plantillas XUL utilizan con llamadas a servicios reales predefinidos, para lo cual sus nombres y métodos deben coincidir con los utilizados en el modelo. Ya que la generación de código no es el objetivo de esta tesis, el lector interesado puede consultar una descripción más detallada de este proceso en [126].

Entorno visual generado para VisMODLE

A partir de la definición anterior, AToM^3 generó el entorno multi-vista de la figura 5.21, donde se muestra la edición de un modelo de comportamiento que pertenece al ejemplo

introducido al comienzo del apartado.

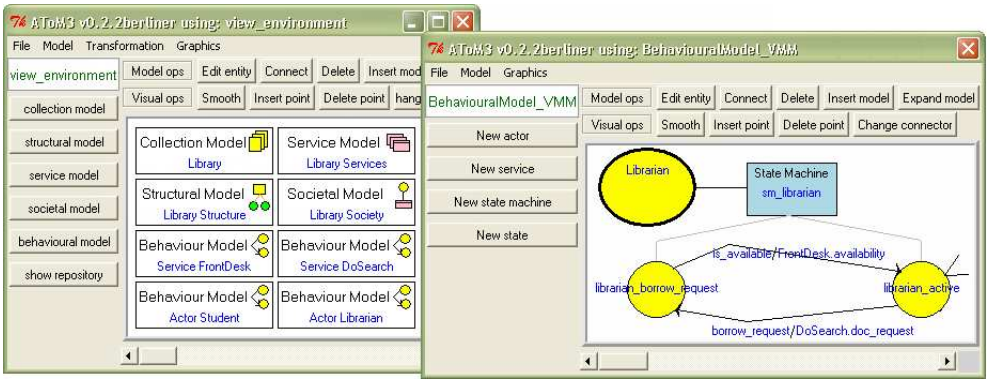


Figura 5.21: *Entorno multi-vista generado para VisMODLE*

La ventana del fondo en la figura 5.22 corresponde a la interfaz generada para el repositorio VisMODLE. El modelo repositorio que incluye es el del ejemplo introductorio, creado automáticamente a partir de las distintas vistas del sistema. El noveno botón de la interfaz permite transformar el modelo repositorio a redes de Petri según el TGTS definido, y muestra el resultado en una nueva ventana. La red de Petri que contiene la ventana en primer plano de la misma figura es la resultante de transformar el repositorio mostrado. De esta manera se puede simular la red de Petri resultante, o realizar análisis distintos de los que proporciona el entorno multi-vista.

Los botones del décimo al decimoquinto de la interfaz del repositorio permiten verificar las distintas propiedades especificadas durante la definición del entorno. Por ejemplo, la figura 5.23 muestra el resultado de ejecutar la propiedad de tipo 2 para el estado `front_desk_borrow` (botón `execute reachability` de la interfaz). El cuadro de diálogo de la derecha permite navegar por los distintos resultados del análisis, esto es, por los distintos caminos de ejecución que llevan al estado analizado.

Finalmente, el último botón de la interfaz ejecuta la consulta predefinida en VisMODLE para obtener la vista derivada con los actores y servicios del repositorio sin un comportamiento definido (ninguno en el ejemplo).

Como se explicó anteriormente, también se ha desarrollado un simulador que permite animar el repositorio. Para ejecutar el simulador hay que acceder al menú desplegable `Transformation` que se encuentra en la parte superior de la interfaz del repositorio, y a continuación seleccionar el fichero que contiene el simulador. La figura 5.24 muestra un momento en la simulación del repositorio. El simulador crea un elemento entrada (cuadrado negro) que apunta al estado actual de cada actor y servicio. En la imagen, el actor `Student` ha pedido un libro y está en estado `student_borrow`. Eso significa que acaba de ejecutarse la transición que va del estado `student_active` al estado `student_borrow`, y su acción ha creado el mensaje `borrow` en la entrada del servicio `FrontDesk` (esquina superior

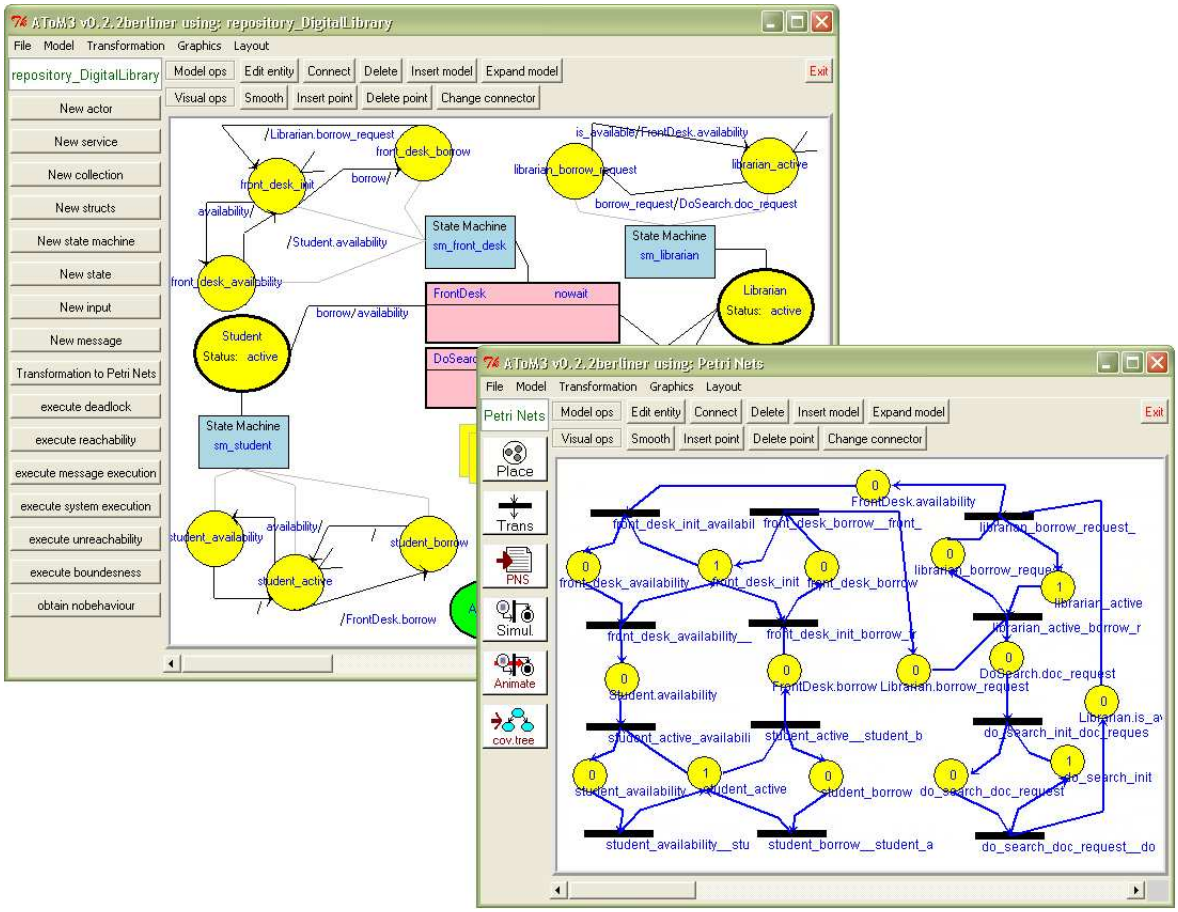


Figura 5.22: *Transformación del repositorio a redes de Petri*

izquierda). El siguiente paso de simulación ejecutaría la transición que va desde el estado actual de `FrontDesk` al estado `front_desk_borrow`, ya que dicha transición requiere un evento `borrow` que se encuentra disponible en la entrada del servicio.

Generación de código desde modelos VisMODLE

Finalmente, para ilustrar la generación de código desde el entorno de modelado VisMODLE, la figura 5.25 muestra la interfaz de usuario generada para el actor `Librarian` de la biblioteca ejemplo. A la derecha, la interfaz incluye en la parte superior la lista de documentos de la biblioteca, obtenida de la colección definida en el modelo de colecciones (véase figura 5.14). Al seleccionar un documento de la lista, éste se visualiza a la derecha utilizando el visor apropiado. Bajo la lista de documentos se muestra el árbol de metadatos de los mismos, cuya estructura se extrae del modelo estructural. Al seleccionar un nodo del árbol de metadatos se activa el cuadro inferior que permite la gestión de metadatos (edición, borrado e inserción).

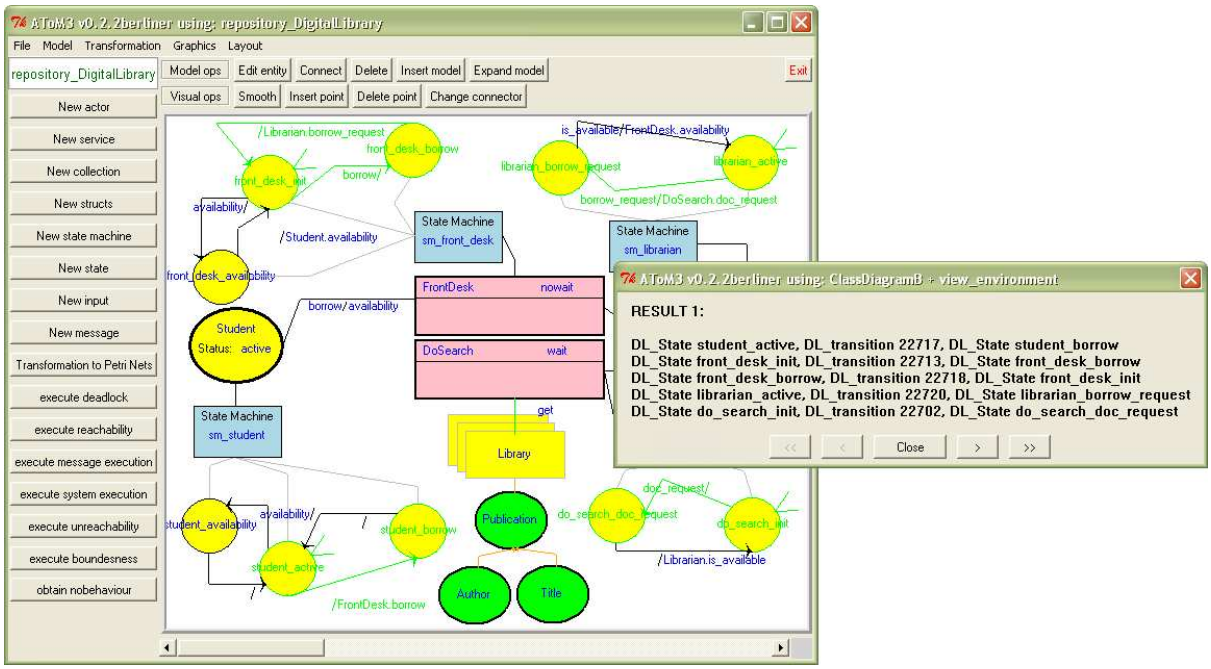


Figura 5.23: *Resultado de verificar una propiedad en el repositorio*

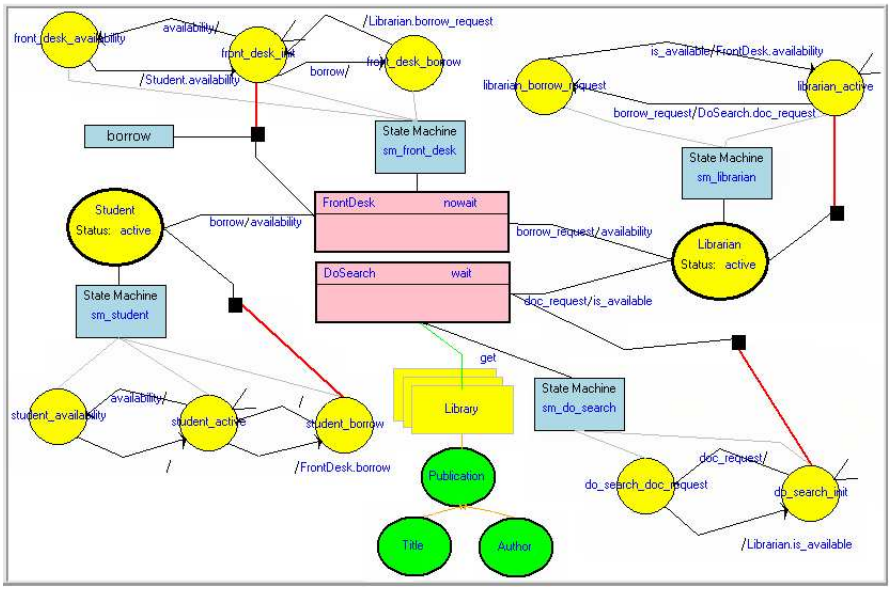


Figura 5.24: *Simulación del repositorio*

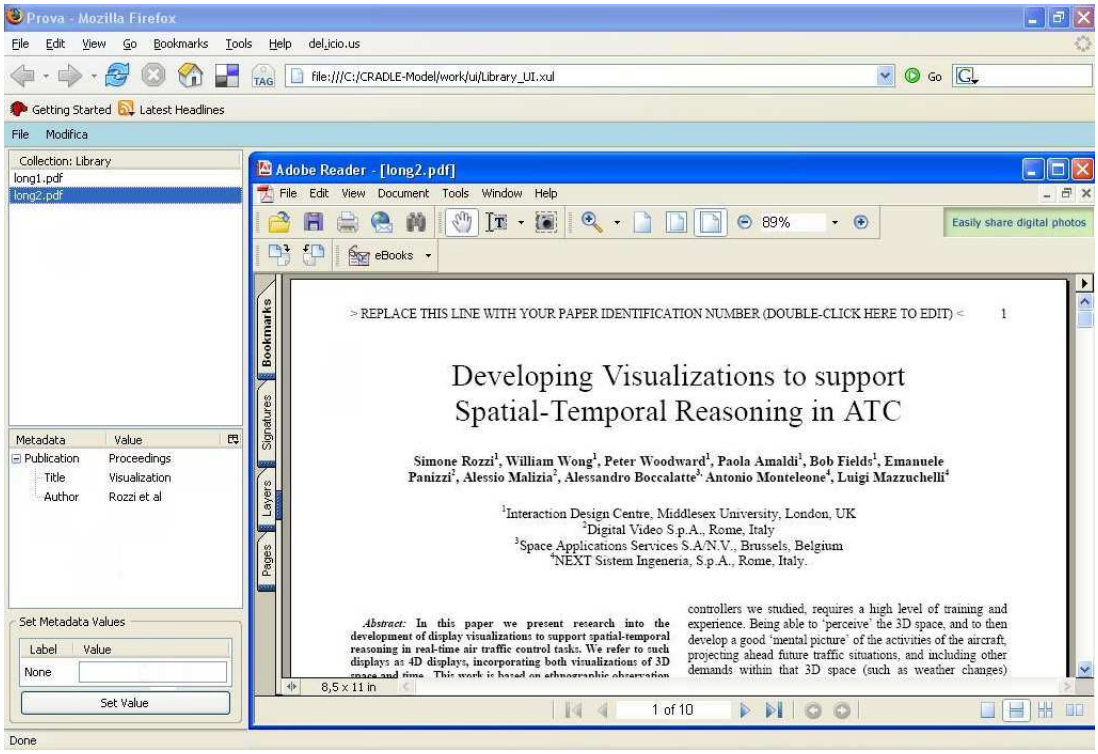


Figura 5.25: *Interfaz de usuario generada desde el ejemplo*

5.1.3. UML

Los apartados anteriores recogían sendos ejemplos de la aplicación del marco multi-vista propuesto en la generación de entornos para dos LVDEs distintos. En este apartado se estudia la generalidad de la propuesta mediante la construcción de un entorno para un pequeño subconjunto del lenguaje de propósito general UML. Al igual que en los casos anteriores, UML está definido mediante un meta-modelo único que recoge todos los conceptos de la sintaxis abstracta del lenguaje. Basándose en este meta-modelo, UML define distintos tipos de diagrama que permiten especificar de manera separada los aspectos dinámicos y estáticos de una aplicación.

La figura 5.26 muestra la definición en AToM^3 de un meta-modelo para una pequeña parte de UML que se tuvo en cuenta para la construcción del entorno. El meta-modelo define clases, asociaciones binarias y relaciones de herencia entre clases, objetos, enlaces entre objetos, máquinas de estados, estados y transiciones entre estados. Como puede verse en la figura, el meta-modelo incluye algunos elementos adicionales que, aunque no forman parte de la definición de UML, se usaron para construir un simulador. Estos elementos auxiliares modelan llamadas a métodos de los objetos a través de un elemento de entrada (*SU_input*) que apunta al estado actual de la máquina de estados de su clasificador. Como puede deducirse de los elementos auxiliares definidos, el simulador que se construyó es similar al presentado en el apartado anterior para la animación de modelos VisMODLE. Por esa razón su definición no se incluye en el presente documento, aunque puede consultarse en [87].

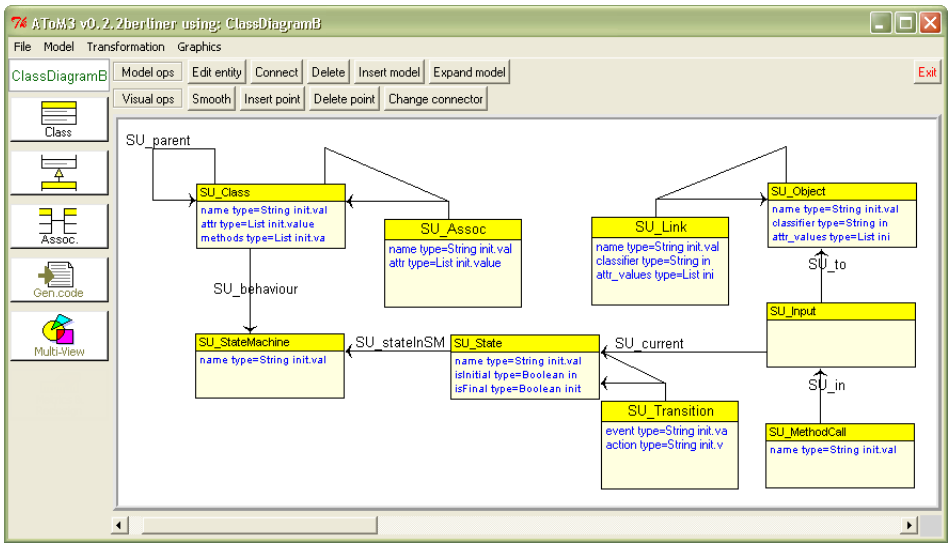


Figura 5.26: *Meta-modelo de un pequeño subconjunto de UML*

Una vez definido el meta-modelo de UML, se especificaron los cuatro puntos de vista que muestra la figura 5.27: un *diagrama de clases* que contiene clases (con nombre, atributos

y métodos) y asociaciones; un *diagrama de jerarquía* que contiene clases (únicamente su nombre) y relaciones de herencia entre ellas[1]; un *diagrama de objetos* con la definición de objetos y enlaces; y un *diagrama de máquinas de estados* para especificar el comportamiento de las clases mediante máquinas de estados. Los elementos auxiliares utilizados por el simulador no aparecen en la definición de ninguno de estos diagramas, sino sólo en el repositorio. En la imagen se muestra la edición del meta-modelo que corresponde a las máquinas de estados.

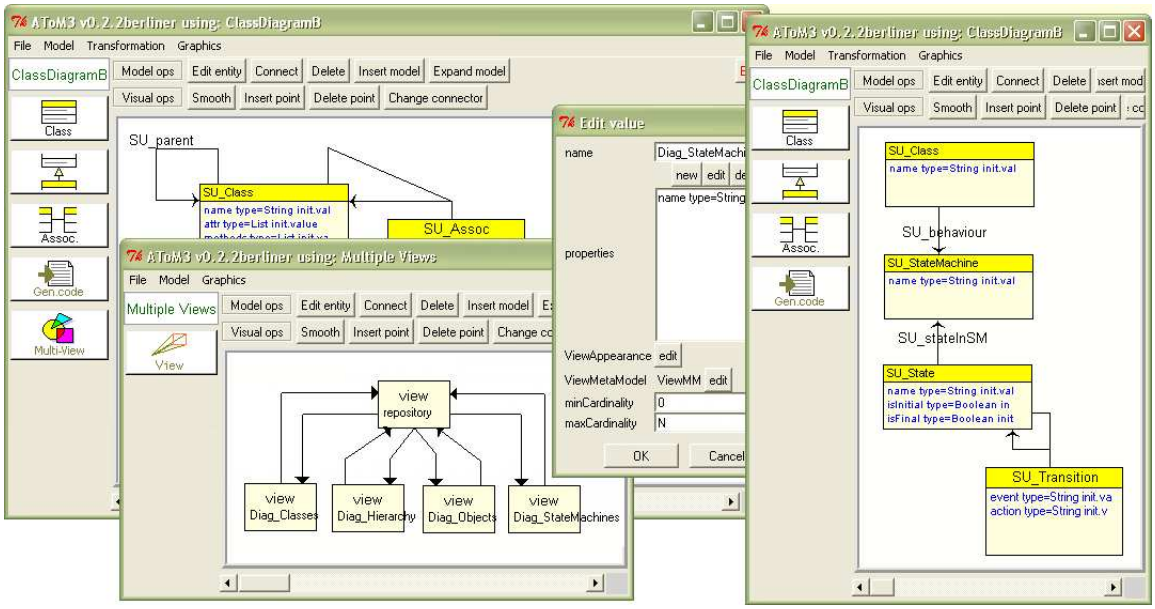


Figura 5.27: *Definición de los puntos de vista de UML*

Debe señalarse que el entorno para UML se definió con una versión preliminar de la herramienta para especificación de lenguajes multi-vista. Esta versión ya generaba automáticamente las relaciones de consistencia entre los puntos de vista y el repositorio, pero no permitía la definición de vistas semánticas ni vistas dirigidas a audiencia. Esto queda patente en la figura 5.27, donde la interfaz de la herramienta de definición de vistas incluye un único botón para crear puntos de vista. En consecuencia no se pudieron incluir mecanismos de análisis ni verificación en la definición del entorno. Sin embargo, la idea de utilizar un enfoque basado en transformación para la verificación de modelos UML fue estudiada por la autora en [83] en el año 2003, y posteriormente defendida en su *Trabajo de Iniciación a la Investigación*. En esos trabajos se proponía una transformación de modelos UML a redes de Petri para su análisis, y se daban una serie de expresiones CTL que permitían verificar diversas propiedades de los modelos realizando *model checking* en el grafo

[1]UML no define un diagrama de jerarquía, sino que las relaciones de herencia se especifican en el diagrama de clases. Sin embargo, en el entorno construido se optó por separar esa información en distintos diagramas.

de cubrimiento de la red. Algunas de esas propiedades permitían, por ejemplo, comprobar si un objeto llegaba a alcanzar cierto estado o si la ejecución del sistema terminaba. La transformación se hacía mediante gramáticas de grafos, y existía una gramática distinta para cada tipo de diagrama. Como cada gramática daba como resultado una red de Petri distinta, tras la transformación había que "componerlas" de tal modo que se obtuviese una única red resultante.

En la actualidad, esos resultados se podrían adaptar para perfeccionar la definición del entorno de modelado para UML presentado. Para ello habría que definir la vista semántica "redes de Petri" y un TGTS desde el repositorio a la vista semántica. El TGTS estaría basado en las transformaciones propuestas en estos trabajos previos, pero ahora habría un único modelo origen (el repositorio) creado automáticamente por las reglas de consistencia, y un único modelo destino resultado de la transformación. Además, los modelos origen y destino se mantendrían limpiamente separados (aunque relacionados) en un solo grafo triple. Cada propiedad a verificar se definiría mediante un método de análisis para el que, además, se podría especificar el mecanismo de anotación. También se podría definir una segunda vista semántica del repositorio UML en el formalismo "redes de Petri coloreadas" mediante el TGTS presentado en [87], para de ese modo poder definir métodos de análisis adicionales.

En conclusión, tras la definición del entorno para UML podemos afirmar que la propuesta para definir lenguajes multi-vista es aplicable no sólo a LVDEs, sino a lenguajes visuales en general. El proceso de generación de reglas de consistencia es válido en ambos casos, y los mecanismos de análisis y anotación de resultados son generales y por tanto aplicables en ambos casos. Esto es bastante lógico si nos ceñimos a una de las diversas acepciones del término LVDE que lo define como "pequeño lenguaje visual restringido a un dominio de aplicación específico". Visto desde esta perspectiva, la generalización del marco a lenguajes visuales generales es sólo una cuestión de escalabilidad, ya que implica manejar meta-modelos más grandes y/o complejos, lo cual no resulta un problema ya que el marco subyacente sigue siendo válido.

Respecto al tipo de propiedades que se pueden querer verificar en un lenguaje de propósito general como UML, también son de carácter más general que las que usualmente se especifican para un LVDE. La razón es que en un LVDE el dominio está muy acotado, y por tanto es posible saber a priori los tipos de análisis y propiedades de interés dentro de ese dominio. En cambio, como los lenguajes de propósito general se pueden usar en áreas muy distintos y su campo de actuación es muy amplio, no es tan fácil determinar qué propiedades resultan interesantes en todos ellos. Por eso estos lenguajes suelen definir mecanismos de análisis muy generales que pueden usarse en un amplio abanico de casos, y es el usuario quien debe saber cuál de ellos usar, cuándo hacerlo y cómo interpretarlos para su sistema concreto.

Entorno visual generado para UML

La figura 5.27 muestra el entorno generado a partir de la definición anterior, el cual proporciona consistencia sintáctica automática entre las distintas vistas del sistema.

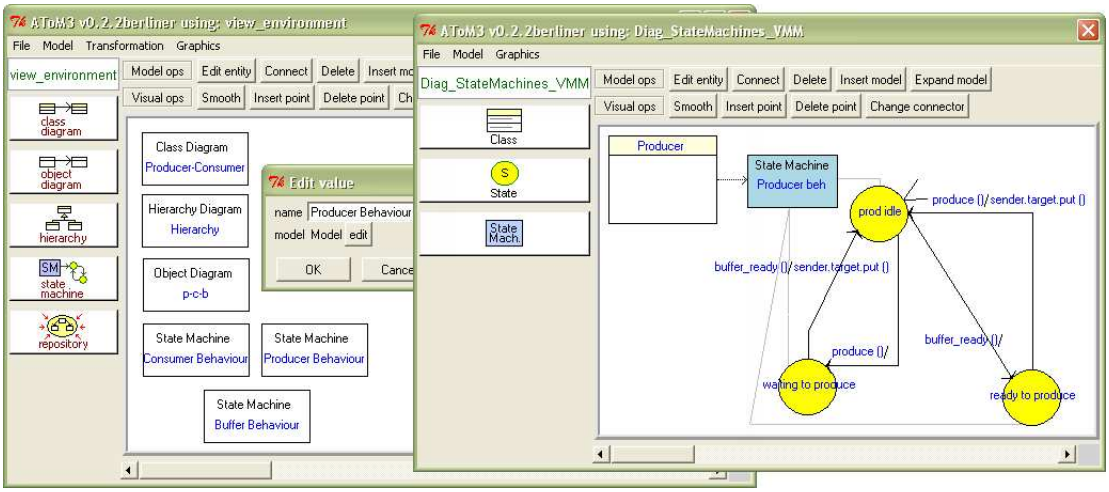


Figura 5.28: *Entorno multi-vista generado para UML*

5.1.4. Otros casos de estudio

Para finalizar, este apartado menciona muy brevemente otros LVDEs multi-vista para los que se ha construido un entorno utilizando la implementación del marco propuesto realizada en AToM^3.

MiCo

MiCo [48] (*Minimal Components*) es un LVDE basado en componentes para el modelado y simulación de eventos discretos. Fue desarrollado inicialmente por Juan de Lara. Posteriormente, la autora de la presente tesis colaboró en la especificación de su meta-modelo, la definición de relaciones sintácticas y semánticas entre las vistas del sistema, y la generación de un entorno visual.

MiCo permite modelar sistemas como un conjunto de componentes con puertos tipados según los eventos que pueden producir y consumir. Define seis puntos de vista distintos: el *diagrama de especificación*, donde se definen los componentes del sistema, qué puertos definen, y su estructura interna en el caso de componentes compuestos; el *diagrama de conexiones*, que incluye restricciones sobre el modo en que los componentes se pueden conectar; el *diagrama de eventos* para definir jerarquías de tipos de eventos; el *diagrama de ejecución*, donde se especifican distintas configuraciones de ejecución hechas de instancias de componentes que ejecutan el comportamiento definido por su tipo; el *diagrama de comportamiento*, que describe el comportamiento de cada componente sencillo (el comportamiento de un componente compuesto está descrito en términos de sus componentes internos); y por último el *diagrama de protocolo*, que especifica el orden en que los eventos se deben enviar o recibir a través de cada puerto, para lo que se usan máquinas de eventos con tiempo que permiten especificar un orden parcial entre los eventos así como protocolos y restricciones temporales adicionales.

El entorno de modelado para MiCo se construyó en AToM^3 definiendo primero su meta-modelo completo, y a continuación sus seis puntos de vista. AToM^3 generó automáticamente las reglas de consistencia sintáctica. Además se incluyeron a mano algunas reglas triples adicionales para verificar propiedades de la semántica estática del lenguaje. Por ejemplo, la regla de la figura 5.29 se añadió a la relación de consistencia generada desde el diagrama de conexiones (parte superior de la regla) al repositorio (parte inferior). Esta regla comprueba el tipo estático de dos puertos antes de conectarlos, ya que un puerto de salida sólo se puede conectar a uno de entrada si este último acepta el tipo de eventos que el puerto de salida genera. Esta información no la incluye el diagrama de conexiones, sino el diagrama de especificación (e internamente el repositorio). Por esta razón la regla realiza la comprobación pertinente en el repositorio y, en caso de no poder realizar la conexión, muestra un mensaje de error y borra la conexión de la vista.

Adicionalmente, para la semántica dinámica hay que comparar los diagramas de com-

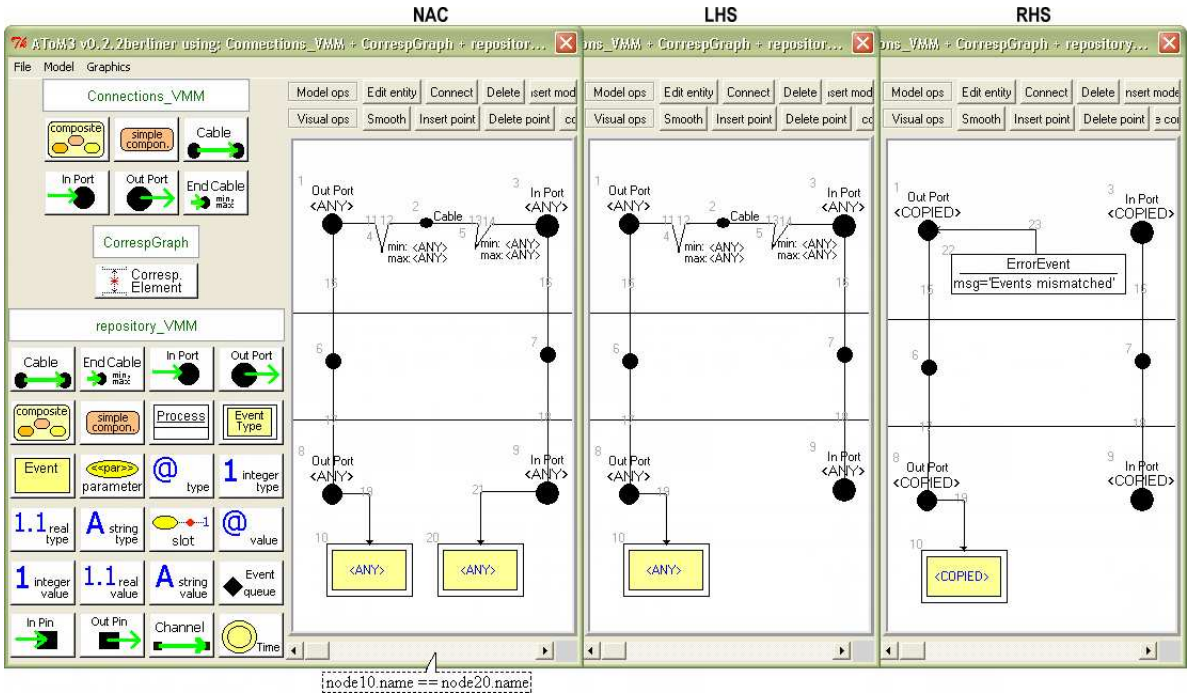


Figura 5.29: *Regla de consistencia de la semántica estática, definida por el diseñador de MiCo*

portamiento de los componentes. Como se ha explicado, en MiCo a los puertos se les asigna una especificación (en forma de autómata) que dice el orden en que los eventos se pueden enviar o recibir. Por tanto, al conectar dos puertos, hay que comprobar que los eventos producidos por el puerto de salida son un sublenguaje de los eventos admitidos por el puerto de entrada. Para ello se definió una gramática de grafos que realiza el "aplanado" de los autómatas asociados a los puertos, y un programa en Python que comprueba la inclusión del lenguaje generado por el autómata del puerto de salida en el lenguaje admitido por el autómata del puerto de entrada. En este caso no se utilizaron vistas semánticas para la transformación ya que el entorno se construyó en una versión previa de AToM^3 que no permitía la definición de este tipo de vistas. La actualización del entorno al nuevo marco aún está pendiente de realizarse.

A partir de la definición anterior se generó el entorno multi-vista que muestra la figura 5.30, donde se muestra la edición del diagrama de especificación de un componente que pertenece al modelado de sistema inalámbrico multimedia.

La figura 5.31 muestra otras vistas que corresponden al modelado del mismo sistema en el entorno generado. El lector interesado puede encontrar una descripción más detallada de MiCo y el entorno generado en [48].

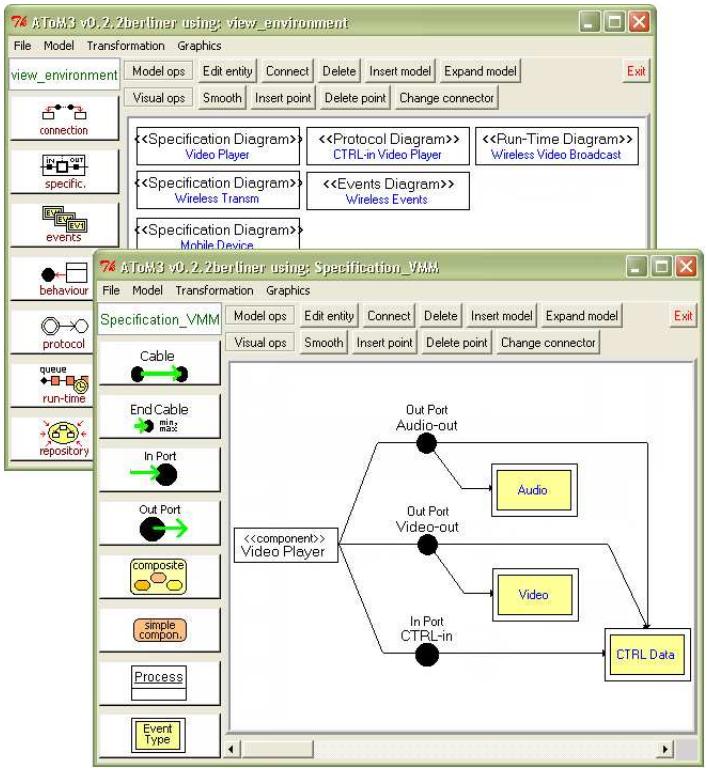
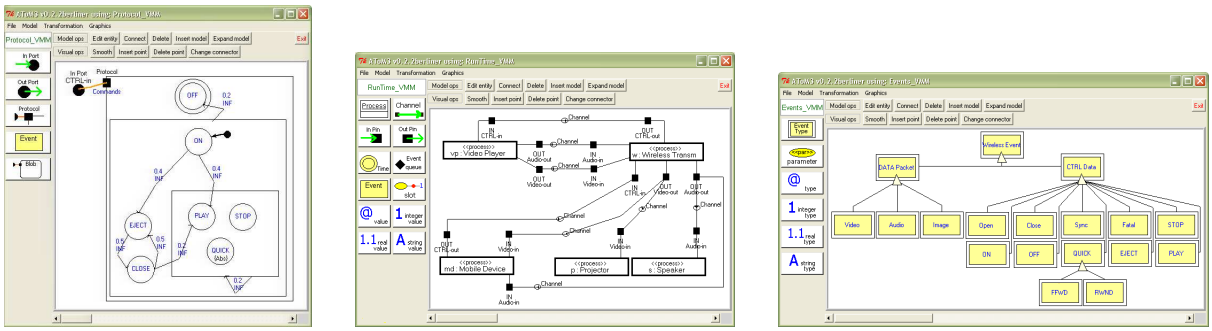


Figura 5.30: *Entorno multi-vista generado para MiCo. Diagrama de especificación*



(a) Diagrama de protocolo (b) Diagrama de ejecución (c) Diagrama de eventos

Figura 5.31: *Vistas del sistema en MiCo*

PRIMA

PRIMA (*Precise Visual Language for Modeling with Agents*) es un LVDE creado por Alexandre Muzy para la construcción de modelos basados en agentes, los cuales se simulan posteriormente transformándolos al lenguaje formal de simulación DEVS [194]. PRIMA propone cuatro tipos de diagrama para describir la estructura del agente, su comporta-

miento, sus instancias y la topología del entorno, respectivamente. El entorno para PRIMA se definió de la manera usual, especificando primero su meta-modelo completo, luego sus puntos de vista, y finalmente generando automáticamente las relaciones de consistencia. A partir de tal definición se generó el entorno multi-vista que muestra la figura 5.32. En la figura se está editando el diagrama de comportamiento de un modelo bioeconómico que estudia las condiciones económicas y biológicas óptimas para la explotación de una población de pescado que lleven a la creación de una reserva marina.

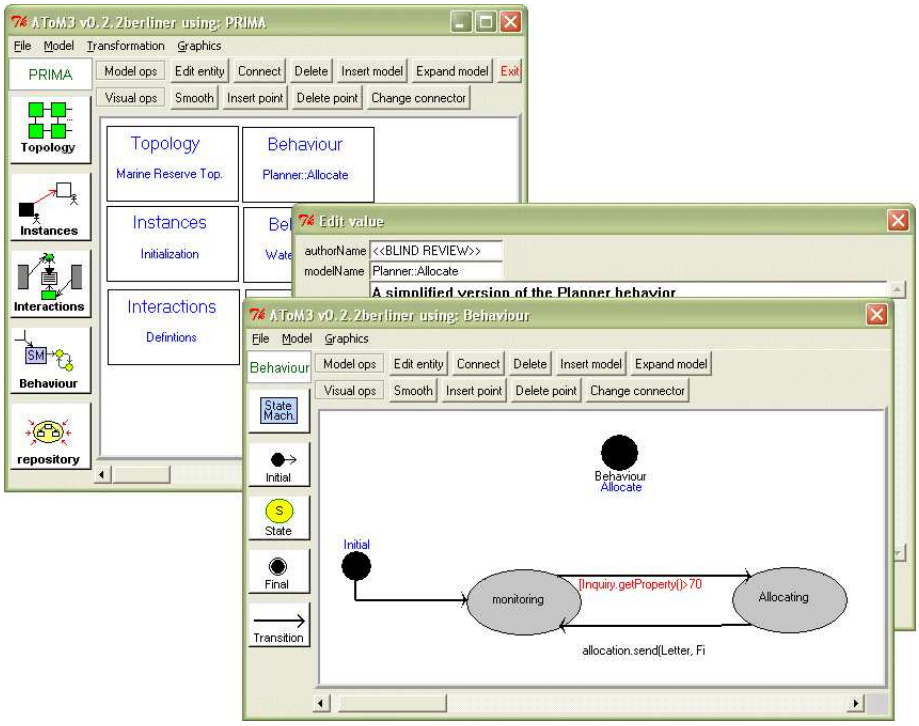
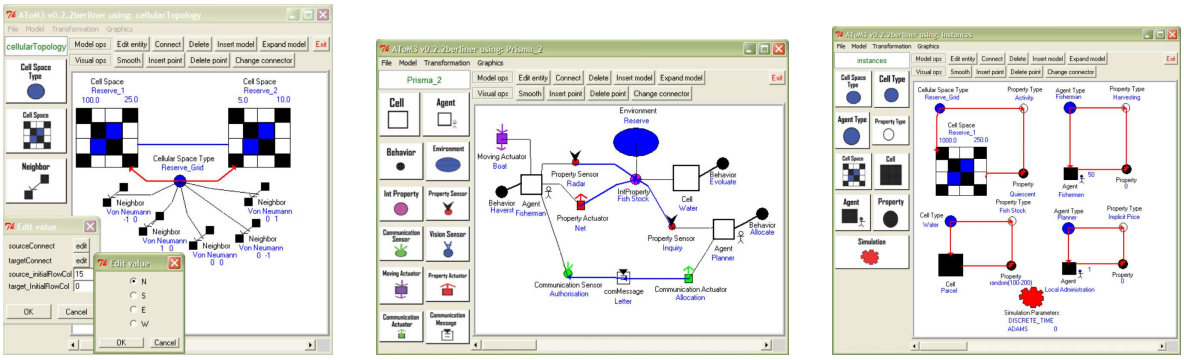


Figura 5.32: *Entorno multi-vista generado para PRIMA. Diagrama de comportamiento*

En la figura 5.33 se muestra un ejemplo de cada uno de los tipos de diagrama restantes que corresponden al modelado del mismo sistema. En concreto la figura muestra la topología de la reserva, las interacciones entre los individuos de la reserva, y las instancias del modelo, respectivamente.



(a) Topología (b) Interacciones (c) Instancias

Figura 5.33: *Vistas del sistema en PRIMA*

5.2. Evaluación empírica del soporte para medidas y rediseños

El objetivo de esta sección es realizar una validación empírica del marco presentado en la sección 4.2. Para ello se presenta un caso de estudio donde SLAMMER se utiliza para la especificación de un conjunto de medidas y acciones sobre un LVDE concreto, a partir de la cual se genera un entorno visual que permite la medición y rediseño de modelos escritos en ese lenguaje. Como lenguaje de estudio se usará Labyrinth [53], que ya se presentó en el apartado 5.1.1 para ilustrar la generación de entornos multi-vista con AToM^3. En esta sección ese entorno se enriquece con un conjunto de mecanismos para la medición de modelos y la ejecución de rediseños específicos del lenguaje que tienen lugar en el repositorio del sistema. Si como resultado de su aplicación el repositorio se modifica, los cambios se propagan desde allí al resto de modelos del sistema.

Antes de continuar debe aclararse que, si bien en esta sección sólo se va a presentar la aplicación de SLAMMER a Labyrinth, en realidad hay que tener en cuenta que éste es un lenguaje multi-vista formado por una familia de LVDEs, cada uno de ellos orientado al modelado de un aspecto distinto en el diseño web. Por esta razón la sección incluye distintos apartados que presentan grupos de medidas y acciones para algunos de esos aspectos o puntos de vista. Como cada uno utiliza conceptos específicos, y además existen medidas y rediseños apropiados para cada uno de ellos, se pueden considerar casos de uso distintos e independientes. De hecho, el amplio espectro de conceptos distintos que permite modelar Labyrinth (navegación, estructura, seguridad, etc.) permitió usarlo a lo largo del proceso de definición de SLAMMER como base de un proceso iterativo de refinamiento en el que la generalidad, capacidad expresiva y compleción de SLAMMER fueron gradualmente validadas. Finalmente, el hecho de presentar como caso de estudio distintos LVDEs que pertenecen a la misma notación, permitirá mostrar la integración de las propuestas realizadas en este documento para el soporte multi-vista y la medición y mejora de la calidad.

Para poder evaluar la riqueza expresiva del LVDE SLAMMER, la sección incluye tres primeros apartados que ilustran su uso mediante la definición de un conjunto de métricas de navegación, métricas de políticas de seguridad y acciones para Labyrinth, respectivamente. En el último caso también se incluye una discusión sobre qué indicadores pueden usarse como disparadores de las acciones, y cómo expresarlo usando SLAMMER. A continuación se incluye un último apartado que muestra la utilización de la herramienta de soporte integrada en AToM^3 para la especificación del modelo SLAMMER presentado en los apartados previos, así como el entorno generado a partir de dicha definición.

5.2.1. Labyrinth

Labyrinth [53, 54] es un LVDE multi-vista para el modelado de sistemas web, cuyo meta-modelo y características ya se presentaron previamente en el apartado 5.1.1. En este apartado se va a ilustrar el uso de SLAMMER usando Labyrinth como caso de estudio. La figura 5.34 muestra el modelo SLAMMER que contiene la definición de las medidas, las acciones y las tareas específicas para Labyrinth, así como sus relaciones y dependencias. El propósito y los atributos necesarios para la configuración de cada elemento de este modelo se explican a continuación.

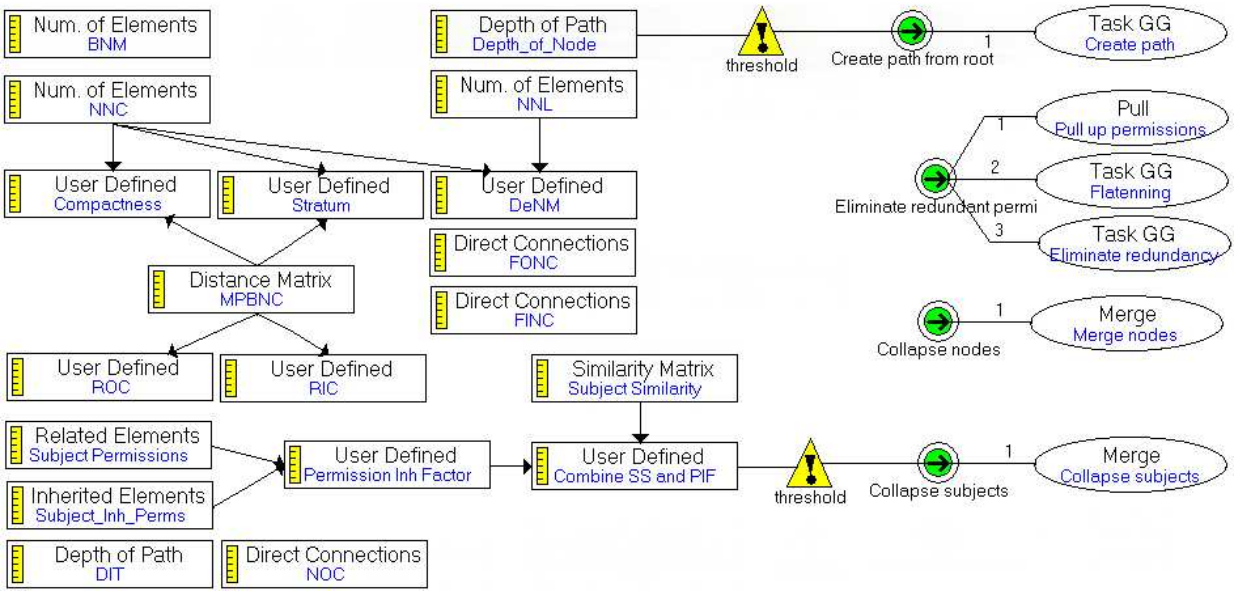


Figura 5.34: *Modelo SLAMMER definido para Labyrinth*

Definición de medidas de navegación

Este subapartado describe aquellas medidas del modelo SLAMMER de la figura 5.34 cuyo propósito es la medición de características de navegación. El conjunto de medidas presentadas es bastante conocido en Ingeniería Web [1, 26]. SLAMMER se utiliza para definirlas y poder aplicarlas posteriormente sobre modelos escritos con Labyrinth. Para cada métrica se mostrará la clase SLAMMER utilizada para su especificación, así como el valor de sus atributos y los patrones visuales utilizados.

El *Número de Contextos de Navegación* (NNC) [1] es una métrica utilizada como indicador del tamaño de un modelo de navegación. Puede servir para detectar árboles de navegación pobremente estructurados, debido quizás a la identificación de requisitos del sistema con destinos de navegación. En el caso concreto de Labyrinth, un contexto de navegación se representa mediante un componente de tipo nodo (clase *NodeComponent*) que

participa en un enlace de navegación (clases *Link* y *Anchor*, más las asociaciones *refersToN* y *source/target*). Así, el NNC puede definirse como una medida SLAMMER de tipo *NumberOfElements* donde el elemento a contar se especifica con el patrón de la figura 5.35. De este modo la medida contará los nodos (simples y compuestos, ya que el atributo *subtypeMatching* toma el valor "sí") que son origen (Y_1) o destino (Y_2) de un enlace de navegación. La salida del patrón es el elemento a contar, es decir, el nodo. Los demás atributos de la medida se muestran en la misma figura. Por ejemplo, la escala son los números enteros y la unidad de medición los "nodos" (es decir, el resultado de la medición serán 0, 4, 6, ... nodos). Finalmente, no se especifica ningún dominio para la medida ya que es orientada a modelo, y por tanto la medición se realiza sobre el modelo completo. Esa es también la razón de que el patrón usado para configurarla no tenga argumentos de entrada.

SLAMMER class: NumberOfElements
Name: NNC
Goal: Tamaño
Domain: –
SubtypeMatching: sí
Scale: [0,N]
Unit: nodos

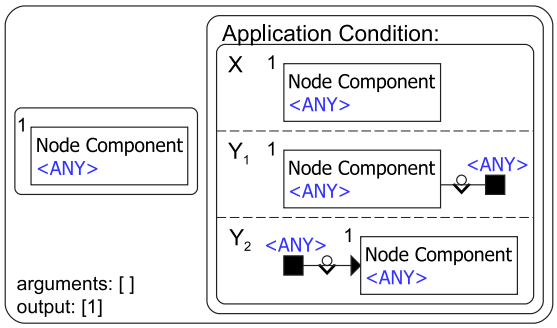


Figura 5.35: *Especificación del "Número de contextos de navegación"*

El *Número de Enlaces de Navegación* (NNL) [1] es otro indicador del tamaño de un modelo de navegación que cuenta cuántos enlaces de navegación hay definidos[2]. Usando SLAMMER podemos definir la métrica NNL para Labyrinth como un objeto de tipo *NumberOfElements* con los atributos que recoge la figura 5.36. El patrón visual utilizado para especificar el elemento a contar, que en este caso son los enlaces existentes entre nodos del sistema, se muestra en la misma figura.

Un tercer indicador del tamaño del modelo de navegación es la *Densidad del Mapa de Navegación* (DeNM) [1], que se calcula como NNC/NNL. Tal y como recoge la figura 5.37, se puede definir como una medida indirecta de tipo *UserDefined* cuya función de medición realice el cociente, para lo cual depende de las métricas NNC y NNL anteriormente definidas. Como se recordará, para definir una dependencia entre dos medidas basta con crear una relación *dependency* entre ellas, lo cual significa que la medida dependiente usará el resultado de la medida de la que depende en su función de medición.

La *Anchura del Mapa de Navegación* (BNM) [1] es una métrica de la usabilidad del primer nivel de navegación. Cuenta el número de contextos que son directamente accesibles

[2]Aunque no es el caso de Labyrinth, existen diversas notaciones para diseño web que permiten especificar enlaces que no son de navegación

SLAMMER class: NumberOfElements
Name: NNL
Goal: Tamaño
Domain: –
SubtypeMatching: sí
Scale: [0,N]
Unit: enlaces

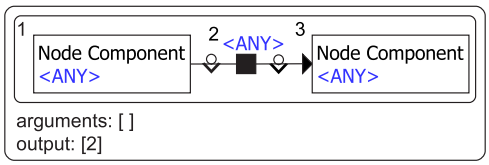


Figura 5.36: *Especificación del "Número de enlaces de navegación"*

SLAMMER class: UserDefined
Name: DeNM
Goal: Tamaño
Domain: –
SubtypeMatching: sí
Scale: [0,N]
Unit: –
Dependencies: NNC, NNL
Calculation: NNC/NNL

Figura 5.37: *Especificación de la "Densidad del mapa de navegación"*

desde el contexto inicial (o página de inicio). Cuanto mayor es su valor más difícil es de usar el contexto inicial, puesto que se le están presentando al usuario muchas posibilidades de navegación al mismo tiempo. Para medir la BNM en modelos Labyrinth se puede usar una medida de tipo *NumberOfElements* configurada con el patrón visual que muestra la figura 5.38. El atributo *isHome* del nodo etiquetado con el número "1" está seleccionado para considerar sólo el contexto inicial. De este modo, al aplicar el patrón, se obtendrán los nodos para los que existe un enlace de navegación directo desde el inicial.

SLAMMER class: NumberOfElements
Name: BNM
Goal: Usabilidad
Domain: –
SubtypeMatching: sí
Scale: [0,N]
Unit: –

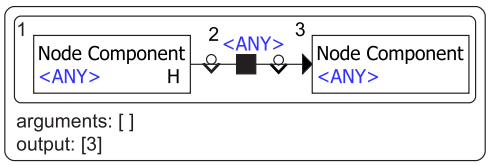


Figura 5.38: *Especificación de la "Anchura del mapa de navegación"*

El *Camino Mínimo entre Contextos de Navegación* (MPBNC) [1] es el número mínimo

de enlaces que hay que navegar desde cada nodo para llegar al resto. Da una medida de la facilidad o usabilidad del mapa de navegación. Para definirla se puede usar la medida orientada a camino *Distance*, que calcula longitudes de caminos. En particular, el tipo de nodo del camino de navegación en Labyrinth (atributo *type*) es "NodeComponent", mientras que la relación que establece un paso en el camino (relación *step*) se expresa mediante el patrón de la figura 5.39. Al aplicar el patrón, el nodo destino de un paso de navegación (la salida del patrón) es el origen del siguiente paso (argumento). Dentro de la escala, un valor igual a -1 implica que no existe un camino entre dos nodos. Nótese que, como se explicó en el capítulo 4, al ser ésta una medida orientada a caminos no hace falta especificar su dominio, sino que éste se calcula implícitamente a partir del atributo *type* y teniendo en cuenta la clasificación de la medida según la dimensión del dominio. Como *Distance* es orientada a grupo se tomaría el dominio ["NodeComponent", "NodeComponent"], permitiendo así calcular la distancia entre cada dos nodos de un modelo.

SLAMMER class: Distance
Name: MPBNC
Goal: Usabilidad
Type: "NodeComponent"
SubtypeMatching: sí
Scale: [-1,N]
Unit: enlaces

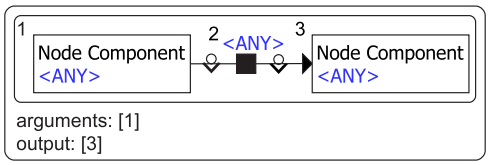


Figura 5.39: *Especificación del "Camino mínimo entre contextos de navegación"*

Las *Centralidades Relativas de Salida* y *de Entrada* (ROC/RIC) [26] miden la facilidad con la que un contexto accede o es accedido desde otros contextos, respectivamente, y suelen usarse para identificar nodos raíz en sistemas con estructura arborescente. Se calculan como las sumas normalizadas de las distancias a/desde cualquier otro contexto. Por tanto, basta con definir una medida indirecta de tipo *UserDefined* que dependa de la métrica MPBNC para calcular la matriz de distancias (véase figura 5.40).

La *Profundidad de un Nodo* (D) [26] es la longitud del camino más corto para llegar al nodo desde el contexto inicial de navegación. Indica la facilidad con la que se puede acceder a un nodo y, en consecuencia, la importancia que tiene para el usuario. A mayor profundidad, mayor dificultad de acceso. Aunque esto puede ser intencional en el caso de nodos de baja relevancia, en otras ocasiones puede ser indicativo de un número de enlaces insuficiente y servir para detectar nodos de información prioritaria que tienen un difícil acceso. Para definirla se puede usar la medida orientada a caminos *DepthOfPath* para el tipo "NodeComponent" y el paso de navegación que recoge el patrón de la figura 5.41. El patrón coincide con el que se usó para definir la métrica MPBNC, ya que en ambos casos lo que se configura es cómo se expresa un paso de navegación en Labyrinth. Sin embargo,

SLAMMER class: UserDefined
Name: ROC/RIC
Goal: Usabilidad
Domain: ["NodeComponent"]
SubtypeMatching: sí
Scale: [0,N]
Unit: –
Dependencies: MPBNC
Calculation: $ROC_i = \frac{\sum_i \sum_j D_{ij}}{\sum_j D_{ij}}$, $RIC_i = \frac{\sum_i \sum_j D_{ij}}{\sum_j D_{ji}}$

Figura 5.40: *Especificación de las "Centralidades relativas de salida y entrada"*

como *DepthOfPath* es orientada a elemento, el dominio que se calcularía implícitamente a partir del atributo *type* sería "NodeComponent" (esto es, la dimensión del dominio es 1). En cualquier caso, esto es algo que en el caso de medidas orientadas a caminos se calcula internamente y no afecta a la hora de configurarlas.

SLAMMER class: DepthOfPath
Name: D
Goal: Usabilidad
Type: "NodeComponent"
SubtypeMatching: sí
Scale: [0,N]
Unit: –

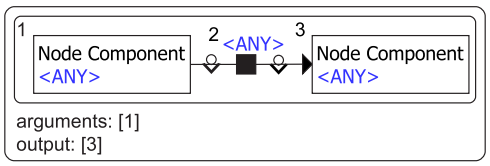


Figura 5.41: *Especificación de la "Profundidad de un nodo"*

Por la misma razón, las métricas *Fan-in* y *Fan-out de un Contexto de Navegación* (FINC/FONC) [1] se pueden definir mediante sendos objetos de tipo *DirectConnections* configurados con ese mismo patrón, tal y como muestra la figura 5.42. Estas métricas cuentan el número de enlaces que llaman a un contexto de navegación o de los que él llama, respectivamente. Dicho de otro modo, lo que cuentan es el número de nodos relacionados con uno dado mediante un camino de longitud 1. Cuanto mayor es este número, mayor es la dependencia entre los nodos y menor su reusabilidad. La medida BNM explicada anteriormente es un caso particular del FONC para el caso del contexto inicial. La figura muestra los atributos y el patrón visual utilizados para configurar el FONC; para configurar el FINC se usarían los mismos, pero intercambiando el argumento y la salida del patrón.

La *Compactibilidad* (C_p) [26] mide el grado de conectividad de un modelo y toma valores en el rango [0,1]. Valores altos significan facilidad para alcanzar cualquier nodo del modelo, quizás provocando la desorientación del usuario, mientras que valores bajos

SLAMMER class: DirectConnections
Name: FINC/FONC
Goal: Reutilización
Type: "NodeComponent"
SubtypeMatching: sí
Scale: [0,N]
Unit: nodos

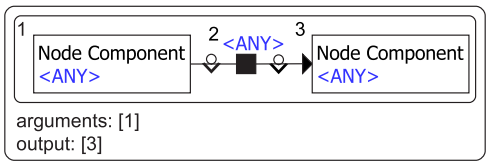


Figura 5.42: *Especificación del "Fan-in y fan-out de un contexto de navegación"*

pueden indicar que el número de enlaces definidos es insuficiente. Por otro lado, el *Stratum* (S) [26] mide el grado de linealidad de un modelo. En este caso un valor alto es indicativo de un sistema lineal que, a pesar de proporcionar una navegación sencilla, puede resultar tedioso de visitar. Las fórmulas necesarias para el cálculo de ambas métricas se muestran en las figuras 5.43 y 5.44 respectivamente, donde n es igual al NNC y las distancias D_{ij} son las que calcula el MPBNC. Por tanto se pueden definir como sendas medidas *UserDefined* con dependencias del NNC y el MPBNC.

SLAMMER class: UserDefined
Name: C_p
Goal: Conectividad
Domain: –
SubtypeMatching: sí
Scale: [0,1]
Unit: –
Dependencies: NNC,MPBNC
Calculation: $\frac{Max-\sum_i \sum_j D_{ij}}{Max-Min}$, $Max = (n^2-n)*k$, $Min = (n^2-n)$

Figura 5.43: *Especificación de la "Compactibilidad"*

Las medidas presentadas en este subapartado miden propiedades genéricas de modelos de navegación que son independientes del tipo de usuario. Sin embargo, en muchos casos, la disponibilidad de un nodo o de un enlace concreto depende de los roles que tenga asignado el usuario que está utilizando el sistema. En esos casos, el modelo de navegación del sistema tendría que refinarse antes de realizar la medición para contemplar sólo los nodos y enlaces a los que el usuario tiene acceso. Con ese objetivo se puede utilizar el marco multi-vista propuesto en este documento para disponer de un repositorio con toda la información del sistema (en este caso relacionando los conceptos de navegación y seguridad en Labyrinth), así como de un mecanismo para la construcción de vistas derivadas (para obtener el modelo de navegación de un usuario particular) sobre las que realizar las mediciones.

SLAMMER class: UserDefined
Name: S
Goal: Linealidad
Domain: –
SubtypeMatching: sí
Scale: [0,1]
Unit: –
Dependencies: NNC,MPBNC
Calculation: $\frac{\sum_i(|\sum_j D_{ij} - \sum_j D_{ji}|)}{lap}$, $lap = n^3/4$ si n par, y $(n^3-n)/4$ si n impar

Figura 5.44: *Especificación del "Stratum"*

Definición de medidas para políticas de seguridad

A continuación se completa el conjunto de métricas definidas para Labyrinth en el modelo SLAMMER de la figura 5.34, esta vez para la medición de propiedades de su control de acceso basado en sujetos (roles y equipos). En particular, ya que Labyrinth permite definir jerarquías de sujetos, se van a presentar un conjunto de métricas que usualmente se utilizan en el dominio de la orientación a objetos [191], pero adaptadas al contexto de la herencia de permisos. Siguiendo con el estilo del subapartado anterior, para cada métrica se mostrará la clase SLAMMER utilizada para su definición, así como el valor de sus atributos y los patrones visuales utilizados para configurarla.

Para empezar, la métrica auxiliar *Permisos de un sujeto* (SP) calcula el número de nodos y contenidos a los que tiene acceso directo un sujeto (es decir, existe una relación de asignación de permiso del sujeto al nodo o contenido). Aunque esta métrica no constituye un indicador útil de ninguna propiedad del sistema, se utilizará en el método de medición de una métrica posterior. Los atributos necesarios para definirla como *RelatedElements* se muestran en la figura 5.45. El patrón visual recibe como entrada el sujeto para el que se toma la medida, y devuelve como resultado los objetos hipermedia (nodos y contenidos) relacionados con el mismo.

SLAMMER class: RelatedElements
Name: SP
Goal: Auxiliar
Domain: "Subject"
SubtypeMatching: sí
Scale: [0,N]
Unit: permisos

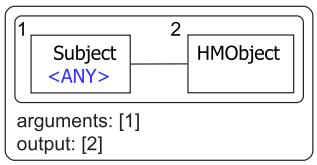


Figura 5.45: *Especificación de los "Permisos de un sujeto"*

Los *Permisos Heredados por un Sujeto* (SIP) se calculan utilizando la medida orientada a caminos *InheritedElements*. Para configurarla hay que definir dos patrones visuales: el primero especifica cómo se expresa un paso en la jerarquía de sujetos de Labyrinth (primer patrón en la figura 5.46), y el segundo especifica el tipo de elementos heredados a través de la jerarquía, que en este caso son objetos hipermedia (segundo patrón en la misma figura). Como puede verse, un solo patrón permite definir distintos tipos de paso en un camino tal y como ocurre en la jerarquía de sujetos de Labyrinth, donde puede haber generalización en el caso de roles (Y_1) y agregación en el caso de equipos (Y_2).

SLAMMER class:
InheritedElements
Name: SIP
Goal: Reutilización
Type: "Subject"
SubtypeMatching: sí
Scale: [0,N]
Unit: permisos

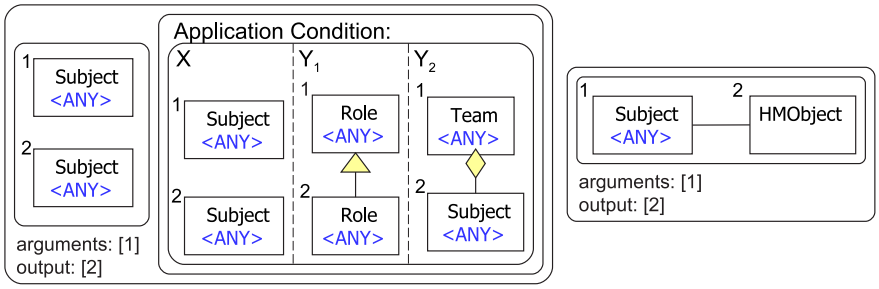


Figura 5.46: *Especificación de los "Permisos heredados por un sujeto"*

El *Factor de Herencia de Permisos* (PIF) calcula el porcentaje de permisos heredados en el sistema. Valores pequeños dentro de la escala pueden indicar un uso deficiente de la herencia y una escasa reutilización de los permisos. Valores altos pueden ser indicativo de una reutilización excesiva que derive en un sistema difícil de mantener, así como de la existencia de roles innecesarios. El PIF es una adaptación de las métricas Factor de Herencia de Atributos y de Métodos (AIF/MIF) que se usan habitualmente en orientación a objetos [191]. Se calcula como la suma de todos los permisos heredados por algún sujeto dividido por la suma de todos los permisos definidos (locales y heredados) por los sujetos. Para definirla usamos una medida *UserDefined* derivada de las dos medidas anteriores, y con los atributos que recoge la figura 5.47.

La *Profundidad del Árbol de Herencia* (DIT) [191] mide la máxima longitud desde cada sujeto al sujeto "raíz" del árbol. Cuanto mayor sea la profundidad del sujeto en el árbol de herencia, mayor será la probabilidad de que esté heredando muchos permisos, haciendo que su comportamiento sea más difícil de prever. Esta métrica se utiliza en orientación a objetos para evaluar la eficiencia y reutilización, aunque también se relaciona con la facilidad de comprensión y el mantenimiento. Para poder usarla sobre la jerarquía de sujetos de Labyrinth, podemos definir una medida de tipo *DepthOfPath* configurada con los atributos que muestra la figura 5.48. Para configurar lo que constituye un paso en el camino de herencia se utiliza el patrón que se muestra en la misma figura, y que coincide con el utilizado en la métrica SIP.

SLAMMER class: UserDefined
Name: PIF
Goal: Reutilización
Domain: –
SubtypeMatching: sí
Scale: [0,1]
Unit: –
Dependencies: SP,SIP
Calculation: $\frac{\sum_i atributos\_heredados(sujeto_i)}{\sum_i atributos\_disponibles(sujeto_i)}$

Figura 5.47: *Especificación del "Factor de herencia de permisos"*

SLAMMER class: DepthOfPath
Name: DIT
Goal: Reutilización
Type: "Subject"
SubtypeMatching: sí
Scale: [0,N]
Unit: –

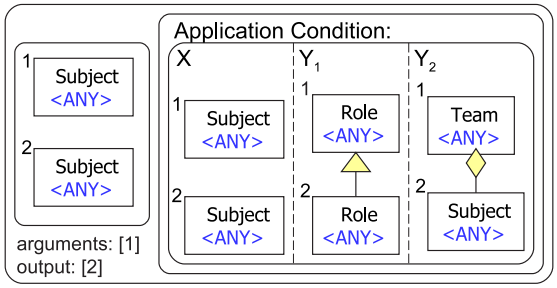


Figura 5.48: *Especificación de la "Profundidad del árbol de herencia"*

El *Número de Hijos* (NOC) [191] es el número de sujetos directamente subordinados a un sujeto en la jerarquía. Si el número es grande la reutilización de permisos es mayor pero también se aumenta la probabilidad de hacer un mal uso de la herencia, para lo que se requerirá aumentar el número de casos de prueba sobre el sujeto y sus descendientes. Puede definirse como una medida *DirectConnections* con los atributos de la figura 5.49 y, de nuevo, el mismo patrón visual para definir un paso en el camino de herencia.

El *Número de Hijos* (NOC) [191] es el número de sujetos directamente subordinados a un sujeto en la jerarquía. Si el número es grande la reutilización de permisos es mayor, pero también aumenta la probabilidad de introducir errores en la política de seguridad derivados de la herencia de permisos entre los descendientes (directos o indirectos) del sujeto. Puede definirse como una medida *DirectConnections* con los atributos de la figura 5.49 y, de nuevo, el mismo patrón visual para definir un paso en el camino de herencia.

Como último ejemplo, la métrica *Similitud de Sujetos* (SS) [167] es un indicador de la cohesión entre sujetos que puede ayudar a detectar redundancias en la política de seguridad. Calcula el parecido de dos sujetos comparando los usuarios a los que están asignados. Esta información sirve para detectar sujetos similares que pueden unirse para minimizar las asignaciones de sujetos a usuarios y así facilitar el mantenimiento. Para definirla se ha

SLAMMER class: DirectConnections
Name: NOC
Goal: Reutilización
Type: "Subject"
SubtypeMatching: sí
Scale: [0,N]
Unit: descendientes

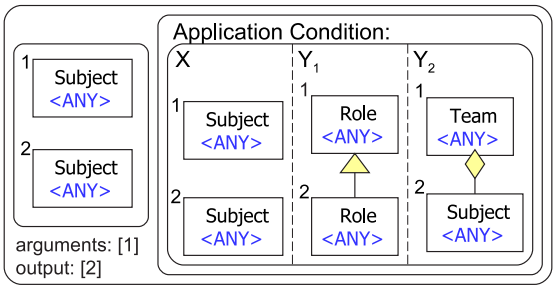


Figura 5.49: *Especificación del "Número de hijos"*

utilizado una *DistanceBasedSimilarity* que devuelve valores en el rango [0,1]. Cuanto menor es el resultado de la medición para dos sujetos, más parecidos son. Para configurarla se han tenido que definir dos propiedades, cada una asociada a uno de los sujetos a comparar. En ambos casos la propiedad expresa cómo se realiza la asignación de sujetos a usuarios en Labyrinth, para lo cual se ha utilizado el patrón que muestra la figura 5.50.

SLAMMER class: DistanceBasedSimilarity
Name: SS
Goal: Cohesión
Domain: ["Subject","Subject"]
SubtypeMatching: sí
Scale: [0,1]
Unit: –

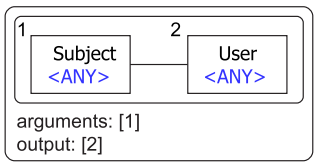


Figura 5.50: *Especificación de la "Similitud de sujetos"*

También se podrían definir medidas con criterios de comparación distintos para establecer la similitud entre sujetos, como por ejemplo estudiando la suma de permisos y atributos definidos y heredan. En ese caso habría que definir una métrica como la anterior, pero especificando como propiedades las relaciones de asignación de permisos y atributos. La medición tendría que realizarse sobre una vista derivada del repositorio donde la jerarquía de herencia se hubiese aplanado. Una opción que evitaría tener que construir tal vista derivada sería ampliar la potencia expresiva de los patrones para permitir condiciones recursivas (al estilo de las usadas en VIATRA2 [17]).

Definición de acciones

Este subapartado ilustra la definición de acciones con SLAMMER, así como su automatización mediante indicadores asociados a medidas. Para ello se muestra la definición de un conjunto de acciones (mostradas previamente en la figura 5.34) que cubre prácticamente

todos los tipos de acciones que SLAMMER permite definir: mediante plantillas, mediante gramáticas de grafos, y combinando tareas de distinto tipo.

Una primera acción de rediseño aplicable a modelos Labyrinth es unir dos componentes de tipo nodo en uno solo que reúna las propiedades de ambos. Esto puede servir para compactar nodos similares con el objetivo de reducir el tamaño del modelo de navegación y facilitar su mantenimiento. Para detectar qué nodos son buenos candidatos para la acción se puede definir una medida *DistanceBasedSimilarity* que evalúe la similitud de los nodos mediante el análisis de los contenidos que incluyen, y tenga un indicador asociado para valores cercanos a 0 que dispare la ejecución de la acción. Esta misma acción se puede usar para fusionar nodos consecutivos con poca información, de tal modo que la navegación resulte más ligera al disminuir el número de enlaces de navegación. En este caso, la detección de los nodos a fusionar debería realizarla el usuario en vez de ir asociada al indicador de una medida, ya que en ocasiones se querrá mantener los nodos separados aunque tengan poca información. Para definir este rediseño con SLAMMER se puede usar una tarea basada en plantilla de tipo *Merge*, y configurada para el tipo "NodeComponent" y el atributo *subtypeMatching* seleccionado. Además, su atributo *rel_duplication* tiene que ser falso para evitar relaciones duplicadas a los mismos contenidos, nodos o sujetos, y su atributo *att_merging* cierto.

Otra acción aplicable a Labyrinth, esta vez especificada mediante gramáticas de grafos, es crear un enlace de navegación desde el nodo inicial del modelo de navegación a otro nodo. La implementación de esta acción con SLAMMER consta de una tarea basada en gramática de grafos con la regla que muestra la figura 5.51, la cual crea el enlace si no existe. La acción puede utilizarse para corregir errores de diseño, creando un enlace para un nodo si no existe un camino que lleve a él, o cuando se necesita navegar por un número alto de enlaces para acceder al mismo. La acción puede automatizarse asociando su ejecución a un indicador apropiado para la métrica *Profundidad de un Nodo* (definido previamente), por ejemplo 0, lo que significaría que no existe un camino de navegación que lleve al nodo. De este modo la creación del nuevo camino sería automática y resolvería el error de diseño.

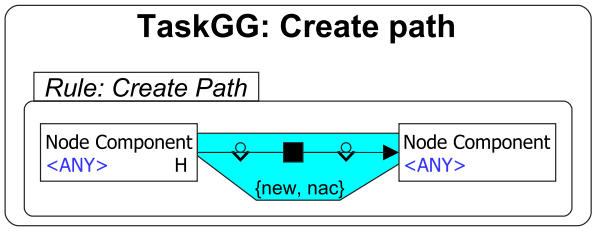
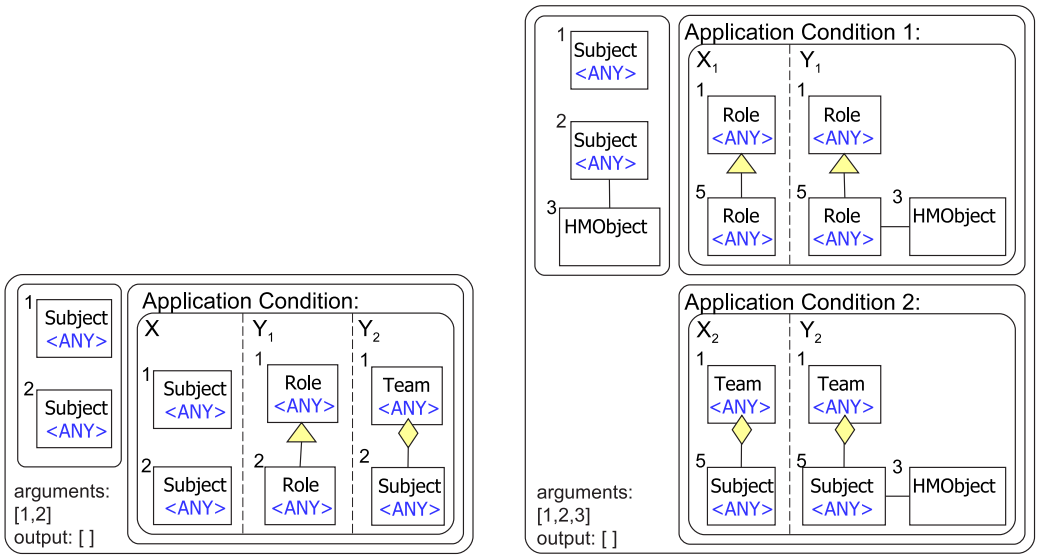


Figura 5.51: *Especificación de una tarea basada en gramáticas de grafos*

Como último ejemplo se ilustra la composición de tareas para llevar a cabo una única acción. En concreto la acción implementa un *refactoring* de modelos que elimina de la

jerarquía de sujetos los permisos redundantes que añaden una complejidad innecesaria al modelo. Para ello, primero mueve a un sujeto los permisos comunes que definen todos sus hijos, y a continuación elimina los permisos asignados a sujetos cuando existe algún ancestro con el mismo permiso, ya que entonces lo heredan de él. La acción está formada por una tarea basada en plantillas y dos tareas basadas en gramáticas de grafos. La tarea basada en plantillas es la que se realiza en primer lugar, y se encarga de subir los permisos definidos en todos los sujetos hijo al padre. Para ello utiliza un objeto *Pull* configurado para el tipo "Subject" (con el atributo *subtypeMatching* seleccionado) y la relación *PA*, que es la que usa Labyrinth para la asignación de permisos. Además debe existir una relación de herencia (por generalización o agregación) entre los sujetos destino y origen, ya que tienen que ser padre e hijo, lo que se indica con el patrón visual de la figura 5.52(a). Finalmente, para mover únicamente aquellos permisos que definen todos los hijos de un sujeto, se restringe la aplicación de la tarea por medio del patrón de la figura 5.52(b). El patrón recibe como entrada la relación a mover y los sujetos fuente y destino. Las condiciones de aplicación comprueban que el permiso que se va a mover existe en todos los hijos del destino, ya sea éste un rol (condición de aplicación X_1–Y_1) o un equipo (condición de aplicación X_2–Y_2).



(a) Patrón *pull_relation* (b) Patrón *constraint*

Figura 5.52: *Patrones visuales para configurar una tarea* Pull

A continuación, la acción elimina los permisos redundantes ejecutando las dos tareas basadas en gramáticas de grafos que muestra la figura 5.53. La tarea *Flattening* "aplana" la jerarquía copiando los permisos de cada sujeto en todos sus hijos. Cada una de las reglas de la tarea se encarga de la propagación de permisos cuando el sujeto padre es un rol o un equipo, respectivamente. Como las reglas se aplican tantas veces como sea posible, los

permisos se propagarán recursivamente hacia abajo por toda la jerarquía. Después, la tarea *Eliminate redundancy* recorre la jerarquía de abajo hacia arriba eliminando los permisos redundantes. Sus tres reglas consideran las tres combinaciones posibles en la jerarquía: un rol padre con un rol hijo, un equipo padre con un rol hijo, o un equipo padre con un equipo hijo[3]. Las condiciones de aplicación no permiten eliminar un permiso si lo tiene asignado algún hijo directo, de tal modo que la eliminación empiece en las hojas del árbol de jerarquía y se vaya propagando hacia arriba nivel a nivel. Al "aplanar" la jerarquía antes de eliminar las redundancias se consigue considerar no sólo los casos en que el permiso está definido por un padre y su hijo, sino también cuando existe un número arbitrario de ancestros entre ellos. Al contrario que las acciones anteriores, ésta no está guiada por un indicador, sino que se aplica sobre toda la jerarquía de sujetos. En cualquier caso eso no constituye un problema ya que el modelado de la seguridad de un sistema no suele implicar un gran número de sujetos.

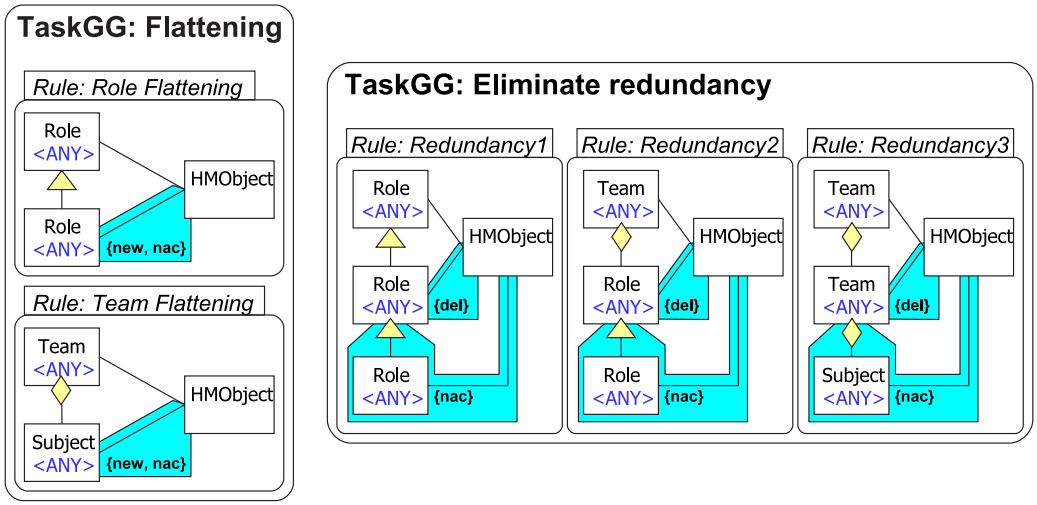


Figura 5.53: *Especificación de dos tareas basadas en gramáticas de grafos*

El procesamiento de una acción a lo largo de cierta estructura recursiva (en el ejemplo anterior el borrado de permisos a lo largo de una jerarquía de sujetos) es muy común en transformación de modelos. En [85] la autora de esta tesis colaboró en la definición de un nuevo tipo de reglas basadas en transformación de grafos que facilita el tratamiento de acciones recursivas, no teniendo que definir reglas adicionales para las fases de pre- y post-procesamiento de la acción (tarea *Flattening* en el ejemplo).

[3]Un equipo no puede generalizar un rol

Generación del entorno visual usando AToM^3

El modelo SLAMMER diseñado en los subapartados previos se utilizó para enriquecer con herramientas de medición y rediseño el entorno multi-vista creado para Labyrinth en el apartado 5.1.1. Para ello, una vez especificado el meta-modelo y los distintos puntos de vista de Labyrinth, se utilizó la herramienta para la especificación de modelos SLAMMER que está integrada en AToM^3 (véase apartado 4.2.4 para una explicación más detallada sobre las prestaciones de la herramienta). La herramienta se utilizó para definir un modelo SLAMMER con un subconjunto de las medidas, indicadores, acciones y tareas expuestas en los subapartados previos. La figura 5.54 muestra este modelo, así como la edición de los atributos de la métrica *Profundidad de un Nodo* que se encuentra en la parte superior izquierda del modelo con el nombre *Depth_of_Node*. El atributo *button* de la métrica está seleccionado, lo que indica que en el entorno generado a partir de este modelo se incluirá un botón para poder realizar la medición. También esta seleccionado el atributo *genReport*, y por tanto el resultado de la medición se grabará en un informe para todos los valores del dominio, ya que además el atributo *report* tiene asignado el valor *complete*. Otro de sus atributos, *step*, tiene como valor el patrón visual que contiene la ventana etiquetada con el número "3". La interfaz para editar el patrón visual contiene todos los elementos del LVDE para el que se está definiendo el entorno visual, y que en este caso son los de meta-modelo Labyrinth.

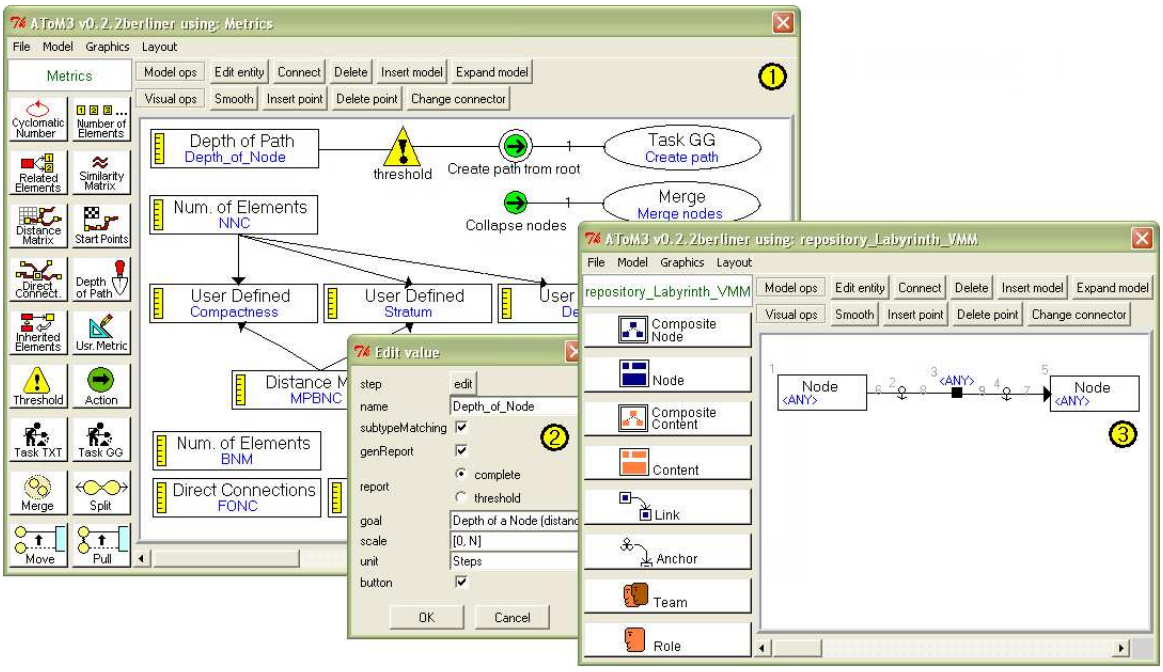


Figura 5.54: *Definición del modelo SLAMMER para Labyrinth*

En el modelo SLAMMER, la métrica *Profundidad de un Nodo* tiene un indicador aso-

ciado para valores iguales a 0 que dispara automáticamente la acción `Create path from root`. Tal y como se explicó anteriormente, la acción está formada por una tarea que crea un enlace directo desde el nodo inicial del modelo de navegación a un nodo dado mediante la regla de la figura 5.51. De ese modo, si el resultado de la medición es 0 para algún nodo (lo que significa que no existe un camino desde el nodo inicial que lleve al nodo) se crea automáticamente un enlace para solucionar este error de diseño.

Por último, como ejemplo de configuración de una métrica *UserDefined*, la figura 5.55 muestra la especificación de la función de medición de la *Densidad del Mapa de Navegación*. Esta métrica tiene como dependencias las métricas NNL y NNC, lo cual significa que el valor de estas últimas puede usarse para el cálculo de la primera (véase figura).

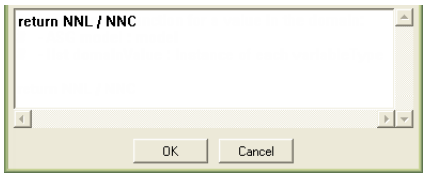


Figura 5.55: *Definición de la función de medición de la métrica DeNM*

A partir de la definición de Labyrinth AToM^3 generó el entorno multi-vista del apartado 5.1.1, pero donde además la interfaz del repositorio contiene botones para poder ejecutar las medidas, acciones y tareas diseñadas en el modelo SLAMMER anterior (si su atributo *button* estaba seleccionado). La nueva interfaz del repositorio para el entorno multi-vista se muestra en la figura 5.56, donde los botones de color amarillo se generaron a partir del modelo SLAMMER. Para realizar una medición o ejecutar una acción basta con pulsar sobre el botón correspondiente.

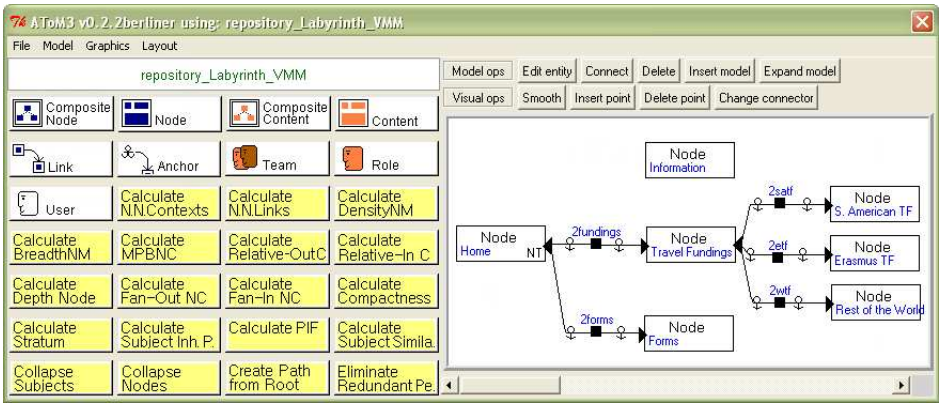


Figura 5.56: *Repositorio enriquecido con botones para la medición y rediseño de modelos*

Por ejemplo, la figura 5.57(a) muestra el informe con el resultado de la medición *Profundidad de un nodo* generado al pulsar el botón `Calculate Depth Node` para el modelo

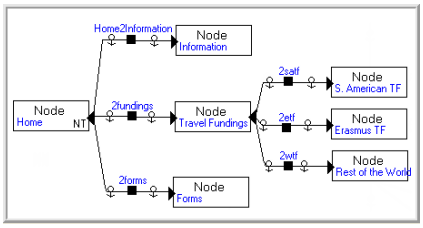
de la figura 5.56. El informe muestra que no existe un camino desde el nodo `Home` al nodo `Information`, ya que éste último tiene una profundidad 0. En cambio se necesita un único paso de navegación para llegar desde el nodo de inicio a los nodos `Travel Fundings` y `Forms`, ya que la profundidad de estos últimos es 1. Como ya se ha explicado, la métrica tiene asociado un indicador que dispara automáticamente una acción para crear un enlace desde el nodo de inicio a aquellos nodos que tienen profundidad 0. Por tanto en este ejemplo la acción se ejecuta para los nodos `Home` e `Information` dando como resultado el modelo de la figura 5.57(b), donde se ha creado un nuevo enlace desde el nodo de inicio al nodo `Information`. Aunque la acción también se ejecuta para el nodo `Home` en ese caso no se crea un enlace que llegue al mismo. La razón es que la regla que se encarga de crear dicho enlace especifica en su LHS que el destino del enlace no puede ser un nodo de inicio.

Metric: D

(complete report)

lb_Node_repository_Labyrinth	Steps
Identifier : Home Type : lb_Node_repository_Labyrinth	0
Identifier : Information Type : lb_Node_repository_Labyrinth	0
Identifier : Travel Fundings Type : lb_Node_repository_Labyrinth	1
Identifier : Forms Type : lb_Node_repository_Labyrinth	1
Identifier : Erasmus TF Type : lb_Node_repository_Labyrinth	2
Identifier : S. American TF Type : lb_Node_repository_Labyrinth	2
Identifier : Rest of the World Type : lb_Node_repository_Labyrinth	2

(a) Informe

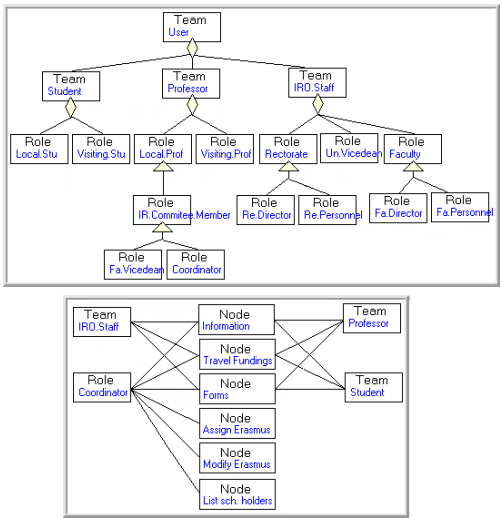


(b) Repositorio

Figura 5.57: *Ejemplo de ejecución de una acción guiada por el valor de un indicador*

La figura 5.58(b) muestra otro informe, esta vez con el resultado de la métrica *Permisos Heredados por un Sujeto* para un sistema que incluye las vistas de la figura 5.58(a). Esos modelos corresponden al diagrama de usuarios y a la tabla de permisos de un sistema web para la administración de una oficina de relaciones internacionales. En el informe aparece reflejado el hecho de que los equipos definidos en el sistema no heredan ningún permiso de sus ancestros. En cambio los roles `Local.Stu` y `Visiting.Stu` heredan tres permisos de su equipo padre `Student`. En este caso la métrica no tiene ningún indicador asociado que dispare la ejecución de una acción. En cambio resulta interesante señalar que para calcular el resultado de la medición se utiliza información de distintas vistas del sistema especificadas por separado, pero que se encuentran almacenadas y relacionadas en el repositorio. Por ello la medición se realiza internamente en el repositorio.

Por último, como ejemplo de acción no guiada por indicadores de medidas y de la propagación de cambios después de un rediseño, se muestra la aplicación del *refactoring* para la eliminación de permisos redundantes sobre el ejemplo anterior que incluye los modelos de



(a) Modelos Labyrinth

Metric: Subject_Inh_Perms

(complete report)

lb_Subject_repository_Labyrinth	permissions
Identifier : User Type : lb_Team_repository_Labyrinth	0
Identifier : Student Type : lb_Team_repository_Labyrinth	0
Identifier : Professor Type : lb_Team_repository_Labyrinth	0
Identifier : IRO.Staff Type : lb_Team_repository_Labyrinth	0
Identifier : Local.Stu Type : lb_Role_repository_Labyrinth	3
Identifier : Visiting.Stu Type : lb_Role_repository_Labyrinth	3
Identifier : Local.Prof Type : lb_Role_repository_Labyrinth	3
Identifier : Visiting.Prof Type : lb_Role_repository_Labyrinth	3
Identifier : IR.Commitee.Member Type : lb_Role_repository_Labyrinth	3

(b) Informe

Figura 5.58: *Informe generado con información proveniente de distintos modelos*

la figura 5.58(a). Para ejecutar el *refactoring* sobre el sistema se pulsa el botón `Eliminate Redundant Pe.` que se encuentra en la interfaz del repositorio. Como se explicó anteriormente, la acción primero sube los permisos definidos por todos los hijos de un sujeto a su padre. Por esta razón los permisos de acceso a los nodos `Information`, `Travel Fundings` y `Forms` se mueven desde los equipos `Student`, `Professor` e `IRO.Staff` al equipo padre `User` que tienen en común. A continuación la acción elimina los permisos redundantes. En el ejemplo los permisos redundantes son los que el rol `Coordinator` define sobre los nodos `Information`, `Travel Fundings` y `Forms`, ya que los hereda del equipo `User`. Al igual que ocurría en el ejemplo anterior, la ejecución de esta acción utiliza información de distintas vistas del sistema que se encuentran relacionadas en el repositorio. Por tanto la acción se aplica al repositorio y el resultado se propaga mediante las reglas de consistencia a las distintas vistas.

La figura 5.59 muestra la tabla de permisos resultante tras aplicar la acción en el repositorio y realizar la propagación de cambios. La tabla sigue conteniendo los equipos `IRO.Staff`, `Professor` y `Student`, aunque no se muestran ya que el rediseño borra todas sus asignaciones de permisos. En cambio, el diagrama de usuarios no se modifica tras ejecutar la acción.

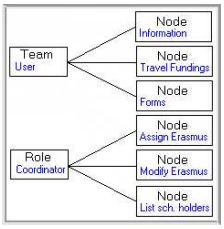


Figura 5.59: *Tabla de acceso tras la ejecución de una acción y posterior propagación de cambios*

5.3. Evaluación analítica del marco

Las secciones previas de este capítulo realizaron una evaluación empírica de la factibilidad y usabilidad del marco propuesto. En esta sección se analiza la originalidad del producto tecnológico resultante (esto es, la extensión de AToM^3 como implementación del marco propuesto) mediante su comparación con las herramientas de meta-modelado presentadas en la sección 3.2.

Los resultados del análisis se resumen en la tabla 5.1. La tabla muestra qué herramientas de meta-modelado definen algún mecanismo para facilitar la especificación de las funcionalidades que muestra la primera columna, de tal modo que éstas se encuentren disponibles en los entornos visuales generados (celdas señaladas, véase sección 3.2 para una descripción más detallada de los mecanismos proporcionados). Si para obtener una funcionalidad sólo se dispone de un lenguaje de programación en el que codificarla a mano desde cero, no se tiene en cuenta en la tabla ya que eso no facilita el trabajo del desarrollador del entorno visual. Por otro lado, sólo se muestran las características relevantes dentro del contexto de esta tesis, y por tanto existen propiedades destacadas de algunas de ellas que no aparecen reflejadas en la misma. También debe tenerse en cuenta que algunas de esas herramientas (por ejemplo DSL Tools, MetaEdit+ y GMF) tienen una implementación comercial frente a otras (como AToM^3, Pounamu o Tiger) que son prototipos académicos.

	AToM^3	**DSL Tools**	**GME**	**GMF**	**MetaEdit+**	**Pounamu**	**Tiger**
múltiples vistas	✓		✓	✓	✓	✓	
consistencia estática	✓		✓	✓	✓	✓	
consistencia dinámica	✓						
comportamiento configurable	✓						
vistas derivadas	✓						
medidas	✓						
rediseños	✓						
generación de informes	∼	✓					
generación de código		✓					
interoperabilidad	baja	media	baja	alta	media	baja	alta

Tabla 5.1: *Comparativa de herramientas de meta-modelado*

Dentro de la tabla, la funcionalidad "comportamiento configurable" se refiere a la posibilidad de poder configurar el comportamiento de los entornos generados. Recuérdese que en la extensión de AToM^3 esto puede realizarse seleccionando diferentes patrones de comportamiento para un LVDE multi-vista (borrado en cascada o conservativo, creación inteligente, etc.), así como modificando los TGTSs de consistencia generados automática-

mente a partir de la descripción del lenguaje. Respecto a la "generación de informes", sólo DSL Tools permite definir informes a medida mediante el uso de plantillas. En AToM^3 los informes con los resultados de una medición están predefinidos. Por último, la "interoperabilidad" se refiere a la capacidad de integración de la herramienta con otras que le proporcionen alguna de las funcionalidades bajo estudio.

5.4. Conclusiones

El capítulo previo presentó un marco para la definición de LVDEs multi-vista que permitía definir mecanismos para la consistencia sintáctica y dinámica de las vistas de un sistema, el análisis y verificación de propiedades, la medición de características de calidad, y el rediseño de modelos. El capítulo se completaba con la implementación de un prototipo que permitía generar entornos visuales a partir de tales definiciones.

En este capítulo se ha realizado una evaluación empírica del marco propuesto utilizando dicho prototipo para generar entornos visuales para LVDEs multi-vista en diversos dominios de aplicación, como son la web, las bibliotecas digitales y la simulación. Para evaluar la genericidad del marco también se generó un entorno para un pequeño subconjunto del lenguaje de propósito general UML.

En todos los casos la definición del entorno visual implicó definir el meta-modelo del lenguaje y el de sus puntos de vista. A partir de esta definición el prototipo generó automáticamente TGTSs para la consistencia sintáctica entre vistas. Los TGTSs se pueden editar posteriormente para modificar el comportamiento generado por defecto. Adicionalmente, la definición de algunos de estos lenguajes se complementó con vistas semánticas para el análisis de propiedades en el dominio semántico redes de Petri, patrones triples para la anotación de resultados, vistas orientadas a audiencia, y modelos SLAMMER con la definición de métricas y rediseños específicos del lenguaje.

A partir de estas definiciones se generaron automáticamente los entornos visuales para los lenguajes. Estos entornos permiten aplicar los mecanismos de consistencia, análisis, medición y rediseño definidos específicamente para el lenguaje sobre los modelos especificados en el entorno.

Capítulo 6

Conclusiones

Los capítulos iniciales de este documento analizaron la necesidad de entornos visuales que no fueran meras herramientas de dibujo y especificación, sino que ayudaran a obtener y controlar la calidad de los modelos especificados con ellas. Con este objetivo se presentaron un conjunto de técnicas formales visuales para facilitar su construcción a partir de una descripción de alto nivel de los lenguajes y de los mecanismos para su análisis, medición y rediseño. Llegados a este punto sólo resta, por tanto, extraer y analizar los resultados obtenidos.

Este capítulo comienza con un resumen de las conclusiones alcanzadas con la realización de esta tesis, y enumera las aportaciones originales a las que ha dado lugar. Para concluir presenta algunas líneas de investigación abiertas que podrían constituir la continuación de este trabajo.

6.1. Conclusiones

El uso de modelos para representar el conocimiento es habitual en muchos dominios. Por ejemplo, en Ingeniería del Software se utilizan para el análisis y diseño de sistemas, o incluso para la generación automática de código a partir de ellos. Para definir los modelos se hace uso de entornos que desempeñan un papel más o menos activo en la tarea: pueden ser meras herramientas de dibujo, o bien incorporar funcionalidades avanzadas para verificar la consistencia sintáctica y semántica de los modelos del sistema, analizarlos, medir sus propiedades, sintetizar código, generar informes, etc. Dado que la calidad exigida a los productos software es cada vez mayor, resulta adecuado aplicar técnicas para asegurar su calidad desde las etapas iniciales del desarrollo, tarea para la que las herramientas del segundo tipo son esenciales. Por tanto, las herramientas de dibujo no son suficientes, sino que se necesitan entornos de modelado con una mayor funcionalidad.

Desarrollar tales entornos no es sencillo y requiere de tiempo y personal experto. Por eso la tendencia actual es facilitar su generación a partir de especificaciones de alto nivel (al estilo del DSDM). Aunque existen muchos enfoques y herramientas para ello, la mayoría genera simples editores de diagramas. Sin embargo, existen ámbitos en el campo de la informática (como el DSDM) que requieren herramientas más funcionales que integren

aspectos para el control de la calidad. Por ese motivo la presente tesis ha propuesto un marco para la generación de entornos de modelado que proporcionen consistencia entre las vistas de un sistema e integren mecanismos de análisis, medición y rediseño adaptados al LVDE usado en el entorno. Los entornos se generan a partir de descripciones formales y visuales (meta-modelado y transformación de grafos). A continuación se incluye una discusión sobre los mecanismos concretos que propone dicho marco, así como del por qué usar TGTSs dentro del mismo.

Sistemas de transformación de grafos triples. Parte del marco propuesto utiliza TGTSs como lenguaje para la transformación de modelos con distintos propósitos. La razón de usarlos es su naturaleza formal y visual, y su capacidad para expresar relaciones entre modelos. Esta capacidad también está presente en el lenguaje QVT, y de hecho los principales resultados presentados (ej. consistencia entre vistas o transformación modelo-a-modelo) serían implementables con la parte visual de QVT sin muchos cambios. No obstante, existen varias razones que llevaron a elegir TGTSs en vez de QVT. En primer lugar, al empezar a trabajar en esta tesis no existía una definición para QVT, y por tanto no era una opción a tener en cuenta. Hoy en día QVT es una realidad, está definido como un estándar, y tiene una mayor riqueza expresiva que los TGTSs. Sin embargo está definido de manera semi-formal, no proporciona técnicas para el análisis de la terminación o confluencia de las transformaciones, y no existen herramientas que soporten su sintaxis visual al completo. Entre las ventajas de los TGTSs están: su naturaleza formal; la existencia de herramientas que lo implementan; que las reglas utilizan la sintaxis concreta del LVDE por ser más intuitiva (QVT utiliza la abstracta); y que el concepto de grafo triple permite definir relaciones entre modelos más complejas que QVT, donde las relaciones son explícitas y pueden contener atributos, y de hecho la relación entre los modelos es un grafo (muy útil para el caso de transformaciones complejas, para definir elementos auxiliares de la transformación o para almacenar información relevante sobre la misma). Su menor expresividad se puede mejorar combinándolos con lenguajes imperativos de control, y algunos trabajos recientes están orientados a definir estructuras de control más avanzadas embebidas en las reglas [58, 60, 85].

La similitud entre TGTSs y QVT [155] ya ha sido señalada por diversos autores, y de hecho existen trabajos que intentan acercarlos o que presentan los TGTSs como una implementación formal (para una parte) de QVT [109]. Siguiendo este razonamiento, no es descabellado considerar que esta tesis está alineada con conceptos presentes en los estándares de la OMG para el modelado de sistemas, pero opta por usar TGTSs en vez de QVT por su carácter formal y la presencia de herramientas.

Finalmente, cabe destacar lo útil que resultó usar AGG para depurar los diversos TGTSs y detectar problemas de confluencia que habían pasado inadvertidos en una

primera versión de los mismos. Si bien es cierto que AGG detecta bastantes pares críticos que no pueden darse en la realidad (lo que se podría mitigar con una herramienta de análisis específica para TGTSs), también es verdad que permite detectar conflictos que se habían pasado por alto.

Generación de entornos multi-vista. Para generar entornos multi-vista se propone definir los puntos de vista del LVDE mediante meta-modelos que son subconjuntos o proyecciones de un meta-modelo global que los relaciona. La consistencia sintáctica entre vistas se obtiene activamente mediante la aplicación de TGTSs que se generan automáticamente a partir de los meta-modelos. Estos TGTSs proporcionan una implementación del patrón Modelo-Vista-Controlador: cuando una vista cambia, un TGTS copia los cambios a un repositorio interno que contiene la suma (colímite) de todas las vistas, desde el cual otro grupo de TGTSs propagan los cambios a las vistas afectadas. En la literatura existen otros enfoques para resolver el problema de la consistencia, siendo ésta una de las pocas funcionalidades estudiadas en la tesis para las que otras herramientas de meta-modelado dan soporte. En comparación con el resto de enfoques, el aquí presentado tiene la ventaja de ser visual y por tanto facilita la modificación de las reglas de consistencia generadas para implementar otros comportamientos. Además, la naturaleza formal de los TGTSs permite su análisis para demostrar si dichas modificaciones afectan o no a la terminación o confluencia de los mismos (tal y como se ha demostrado para las reglas generadas por defecto). También es relevante el hecho de poder configurar el funcionamiento del entorno visual según distintos paradigmas de comportamiento, lo que se consigue mediante la generación del conjunto de reglas triples apropiado en cada caso.

Respecto a la definición del LVDE multi-vista en sí, la implementación en otras herramientas de meta-modelado suele realizarse en la dirección contraria a la propuesta: en vez de partir de la definición de un meta-modelo único del que extraer los puntos de vista, parten de la definición de los puntos de vista y construyen su colímite a través de las clases con el mismo nombre, o definiendo explícitamente referencias entre los elementos equivalentes de los puntos de vista. Cada enfoque tiene sus ventajas e inconvenientes. Por un lado, el partir de un meta-modelo único permite definir en él elementos auxiliares que no pertenecen al lenguaje y por tanto no se usan en ningún punto de vista, pero que pueden ser útiles para otros propósitos como la simulación o el análisis. El capítulo 5 de este documento presentó un ejemplo práctico con el entorno generado para el lenguaje VisMODLE, cuyo repositorio contenía elementos auxiliares para la animación del repositorio que no se incluían en ninguno de los puntos de vista del lenguaje. Como desventaja, este enfoque requiere algo más de trabajo para el desarrollador del entorno visual, pues aparte de los puntos de vista debe definir el meta-modelo global. Esto se atenúa en la implementación realizada,

donde el meta-modelo global se copia a los puntos de vista y lo único que hay que hacer es borrar elementos de estos últimos.

Por último, el que los puntos de vista sean subconjuntos del meta-modelo global es una limitación de la implementación realizada que facilita la generación de las reglas, pero no es una restricción del enfoque. De hecho, en ocasiones puede ser interesante añadir elementos auxiliares en las vistas (ej. para simulación o análisis) que no queremos incluir en el repositorio único. Desde esta perspectiva, las vistas semánticas pueden considerarse un punto de vista especial que no es de especificación, sino de análisis, y cuyos elementos no pertenecen al lenguaje. Otra aplicación donde esto puede ser útil es la definición de LVDEs con sintaxis abstracta y concreta muy distintas (ej. hay elementos de la sintaxis abstracta sin representación gráfica, o viceversa, varios elementos de la sintaxis abstracta se representan como uno solo en la concreta, etc.). En ese caso las vistas podrían contener la sintaxis concreta del lenguaje, el repositorio su sintaxis abstracta, y los TGTSs construirían la sintaxis abstracta de los modelos mientras los usuarios trabajan con la concreta. En consecuencia se podrían tener relaciones arbitrarias entre las dos sintaxis [86].

Cabe señalar que aún no existe una propuesta definitiva para la definición de vistas por parte de la OMG dentro de su lenguaje QVT.

Vistas derivadas y orientadas a audiencia. En ambos casos se ha propuesto el uso de patrones visuales para expresar de forma declarativa y visual consultas sobre modelos. Su principal aportación es que el resultado de la consulta se construye ejecutando un TGTS generado automáticamente a partir del patrón, el cual proporciona mecanismos de sincronización entre el modelo base y la vista derivada. Cuando se definieron se tomó la decisión de no proporcionar sincronización bidireccional, sino sólo propagación de cambios desde el modelo base al resultante de la consulta. A pesar de ello sería fácil generar un TGTSs para propagar cambios en la otra dirección. De esa forma los cambios en el resultado de una consulta se propagarían al modelo base mediante el nuevo TGTS, y desde allí al resto de vistas mediante las reglas de consistencia.

Existen diversos aspectos de los patrones visuales de consulta que podrían mejorarse. En primer lugar habría que dotarles de una mayor riqueza expresiva, ya que no permiten definir consultas muy complejas, sólo condiciones positivas o negativas independientes que afectan a elementos sencillos. En ese sentido, otros lenguajes como OCL son más ricos (aunque éste en concreto no proporciona mecanismos de sincronización). Por otro lado, se ha detectado que los patrones resultan más intuitivos para definir vistas orientadas a audiencia que vistas derivadas. Esto se debe a que los elementos a obtener se tienen que especificar mediante un meta-modelo, el cual

no tiene por qué conocer el usuario final del entorno de modelado. También es cierto que ése es el enfoque que siguen lenguajes de consulta tan conocidos como SQL, donde el constructor de la consulta debe conocer el esquema de la base de datos. Una posible solución sería que, en vez de especificar los elementos a obtener mediante un meta-modelo, éstos se indicaran usando su sintaxis concreta y a partir de ella derivar internamente el meta-modelo implícito.

Integración de mecanismos de análisis. Para integrar mecanismos de análisis en los entornos generados mediante meta-modelado, se ha sugerido complementar la definición de los LVDEs con vistas semánticas. Éstas son vistas del sistema de "sólo-lectura" expresadas en un formalismo que permite el análisis, simulación o verificación de propiedades de interés. Para ello, el desarrollador del entorno debe definir un TGTSs que transforme la vista a analizar al formalismo seleccionado, el mecanismo de análisis de la propiedad a verificar, y el modo de anotar el resultado obtenido sobre el modelo original mediante patrones triples.

Debe señalarse que la idea de analizar y verificar sistemas mediante su transformación a otros dominios semánticos no es nueva, sino que ha demostrado ser útil en muchos campos de aplicación [92, 121, 175, 184, 186, 192]. En cambio, la definición de mecanismos de anotación generales no se ha tenido mucho en cuenta, a pesar de que esa es la única forma de que el usuario del entorno final sepa el significado de lo que está analizando. La utilización de TGTSs para realizar la transformación permite crear correspondencias entre los modelos origen y destino, las cuales se utilizan para realizar la posterior anotación de resultados. Para especificar la anotación resulta bastante natural usar patrones triples indicando lo que se quiere anotar y cómo se quiere notar.

Integración de mecanismos de medición y rediseño. Para cubrir este objetivo se ha definido un LVDE denominado SLAMMER que permite definir métricas y rediseños para un LVDE dado. El enfoque tiene la ventaja de ser independiente del dominio y del lenguaje, y fácilmente configurable para LVDEs arbitrarios. El lenguaje contiene un conjunto de métricas predefinidas configurables para un LVDE mediante patrones visuales, y permite crear métricas nuevas, componer métricas para definir otras más complejas y ejecutar automáticamente rediseños con objeto de mejorar el valor de una métrica. Los rediseños se especifican bien textualmente, mediante transformación de grafos, o configurando una plantilla de las proporcionadas para sencillas tareas predefinidas.

El enfoque desarrollado para definir métricas genéricas mejora otros existentes en la literatura porque desacopla el meta-modelo de las métricas y el meta-modelo que contiene los conceptos del LVDE al que se aplica. De este modo las métricas de

SLAMMER son totalmente independientes del dominio y del lenguaje, y pueden aplicarse a cualquier LVDE. Por el contrario, el estado del arte mostró que las propuestas existentes suelen estar orientadas a un dominio concreto. El uso de patrones permite un alto nivel de abstracción y reusabilidad (un mismo patrón utilizado en distintas métricas medirá propiedades muy distintas) y hace fácil la configuración de métricas de manera gráfica y declarativa. Además, SLAMMER incluye entidades para modelar acciones y su relación con métricas, haciéndolo más completo para el remodelado de software.

Aunque el número de métricas predefinidas en SLAMMER es pequeño, su aplicación al lenguaje Labyrinth demuestra que cubren la medición de bastantes características de un modelo de diseño. Por ejemplo, el tamaño puede ser una medida de la complejidad del sistema, las medidas de similitud o elementos relacionados se pueden usar como medidas de la cohesión, etc. Todas estas métricas están pensadas para ser calculadas automáticamente a partir de los modelos, esto es, SLAMMER no proporciona soporte para métricas de tipo subjetivo. Por otro lado, SLAMMER permite la definición de nuevas métricas mediante composición de otras métricas o codificando la función de medición. Aquí una posible mejora sería poder utilizar patrones para medir propiedades de los modelos, cuyo resultado pudiese usarse en el código especificado.

Respecto a las acciones, su división en tareas dentro de SLAMMER fomenta la reutilización. Sin embargo, la ejecución de tareas secuencial parece insuficiente, y por tanto sería necesario definir un lenguaje de control de flujo más expresivo que incluyera bucles, ejecución condicional, etc. Eso asemejaría el lenguaje propuesto a un lenguaje de acción semántica [182]. Respecto a la decisión de utilizar transformación de grafos para la especificación de tareas, se tomó debido a su naturaleza formal y a la existencia de diversos trabajos que avalan su utilidad para el *refactoring* de modelos [58, 104, 133]. Finalmente, el conjunto de tareas predefinidas en SLAMMER (que son las basadas en plantillas) puede ahorrar trabajo al desarrollador del entorno visual, aunque debería ampliarse para permitir manipulaciones más complejas.

La factibilidad del marco se ha demostrado mediante la construcción de un prototipo que toma como base la herramienta de meta-modelado AToM^3, y permite la generación de entornos visuales para LVDEs multi-vista con mecanismos de consistencia entre vistas, análisis, medición y rediseño específicos del lenguaje. Su utilidad ha quedado patente mediante la construcción de entornos de modelado para LVDEs en distintos ámbitos.

6.2. Aportaciones

La principal aportación de esta tesis ha consistido en definir un marco formal y visual para la generación de entornos para LVDEs multi-vista con capacidad para controlar la calidad de los modelos descritos con tales lenguajes (y, en consecuencia, la calidad del sistema que definen). Los entornos se generan automáticamente a partir de una descripción de alto nivel del lenguaje y de los mecanismos de control de calidad. Entre estos últimos es posible definir mecanismos de consistencia sintáctica y semántica entre vistas, métodos formales ocultos para el análisis del sistema (y donde los resultados se expresan en términos del lenguaje visual para el que se genera el entorno), vistas de "sólo-lectura" orientadas a un tipo concreto de audiencia con la información del sistema que le interesa, métricas y rediseños específicos para el lenguaje, y detección de "malos olores" que proponen la ejecución de acciones predefinidas para su eliminación.

El objetivo final de las técnicas presentadas es ayudar a los desarrolladores a construir entornos visuales en menos tiempo y con un menor coste de mantenimiento. Además, los entornos generados proporcionan gran parte de la funcionalidad que las herramientas CASE desarrolladas de manera tradicional (esto es, codificadas a mano) poseen, ayudando a los usuarios no sólo a dibujar los modelos del diseño de sus sistemas, sino también a verificar su corrección y a controlar y mejorar algunas de sus características. Esto hace factible la integración de tales herramientas en procesos de desarrollo software para el modelado de sistemas en dominios de aplicación reales.

A continuación se detallan los resultados obtenidos más relevantes. Para cada uno se indica, además, qué objetivos de los planteados en el capítulo inicial cubren:

- Formalización de los sistemas de transformación de grafos triples tipados atribuidos (véase apéndices C y D), que constituyen un lenguaje formal, visual y declarativo especialmente adecuado (entre otras cosas) para la transformación modelo-a-modelo. En ese caso los modelos origen y destino de la transformación son dos de los grafos del grafo triple, mientras que el tercero incluye relaciones entre los primeros. Al ser un lenguaje formal es posible verificar propiedades de las transformaciones, como su confluencia o terminación. Aunque la formalización de estos sistemas no resuelve directamente ninguno de los objetivos planteados inicialmente, se utiliza como base para la resolución de algunos de ellos.

- Formalización de los LVDEs multi-vista, lo que incluye mecanismos adecuados para la consistencia sintáctica entre vistas. En concreto, el LVDE se define mediante un meta-modelo único del cual los puntos de vista son subconjuntos. La intersección entre cada dos puntos de vista identifica las interdependencias. A partir de esta definición se generan automáticamente un conjunto de sistemas de transformación de

grafos triples (para distintos paradigmas de comportamiento) cuya ejecución garantiza la consistencia entre vistas del sistema. Los sistemas de transformación se pueden modificar o incrementar con nuevas reglas que implementen restricciones semánticas adicionales. *(Objetivos O1.1, O1.2)*

- Definición de un mecanismo para la integración de métodos formales ocultos en entornos de modelado generados mediante meta-modelado, y que permitan verificar la semántica dinámica del sistema y otras propiedades de interés. El mecanismo se basa en la transformación de los modelos del sistema a un dominio semántico para su análisis posterior. Aunque el enfoque no es novedoso (salvo por el lenguaje de transformación utilizado, que son sistemas de transformación de grafos triples), sí que son aportaciones relevantes:

  - el uso de patrones visuales triples para anotar los resultados del análisis en el dominio semántico a la notación original, que es la que conoce el usuario del entorno.
  - la integración de tales técnicas durante la definición de un entorno visual, de tal modo que cuando éste se genera integra los mecanismos de análisis definidos que son adecuados para el LVDE concreto, y cuyos resultados se muestran en términos del LVDE.
  - la flexibilidad de poder tener en un entorno de modelado distintos dominios semánticos, cada uno para realizar distintos tipos de análisis.

  (Objetivos O1.3, O1.4)

- Definición de un lenguaje de consultas visual para modelos. El lenguaje también se puede usar para definir puntos de vista de un LVDE que son de "sólo-lectura", y que permiten obtener una vista con información parcial del sistema que puede resultar de interés para determinado tipo de audiencia. En ambos casos la consulta se expresa mediante patrones visuales a partir de los cuales se genera un sistema de transformación de grafos triple para la sincronización de los modelos base y derivado frente a posteriores modificaciones del primero. *(Objetivo O1.5)*

- Definición de un LVDE para facilitar la definición de métricas y rediseños específicos del dominio y del lenguaje, donde además los primeros se pueden utilizar para guiar la ejecución de los segundos. Los modelos implementados con tal lenguaje se utilizan, además, para generar automáticamente herramientas de medición y rediseño que se integran en los entornos de modelado generados. *(Objetivos O2.1, O2.2, O2.3, O2.4)*

- Implementación de prototipos del marco propuesto, así como su aplicación en la generación de herramientas de modelado para LVDEs en distintos dominios. *(Objetivos*

O1.6, O2.5)

Las aportaciones enumeradas tienen impacto en la comunidad de Lenguajes Visuales por dos razones. En primer lugar, las técnicas propuestas (transformación de grafos, patrones y LVDEs) son un ejemplo práctico de utilización de técnicas visuales para la resolución de problemas, en este caso la generación de herramientas CASE. En segundo lugar, las herramientas generadas con el enfoque propuesto integran mecanismos para controlar la calidad de modelos expresados mediante notaciones visuales (de dominio específico), mecanismos que además capturan conocimiento del lenguaje y su dominio de aplicación al estar definidos expresamente para ellos.

Las aportaciones de la tesis también son relevantes en la comunidad de transformación de grafos, ya que incluyen la formalización de los sistemas de transformación de grafos triples como base (formal) para la transformación modelo-a-modelo. Dentro del contexto de esta tesis su uso se aplica a la definición de mecanismos de consistencia entre vistas de un sistema, el análisis basado en métodos formales ocultos y la sincronización entre un modelo base y uno derivado.

Por último, facilitar la creación rápida de herramientas de modelado con funcionalidades añadidas como el análisis, medición o rediseño de modelos específicos del lenguaje puede beneficiar a diversas comunidades, entre las cuales destacan las dedicadas al desarrollo dirigido por modelos por necesitar de tales herramientas para desplegar todo su potencial.

6.3. Líneas de trabajo futuro

La realización de este trabajo abre la puerta a líneas de trabajo futuro que comprenden extensiones a lo aquí presentado, así como otras líneas de investigación complementarias. A continuación se describen las más relevantes.

6.3.1. Extensiones al trabajo realizado

Algunas ampliaciones al trabajo presentado, realizables a corto o medio plazo, son las siguientes:

- Integrar mecanismos de análisis para TGTSs en AToM^3. De este modo se evitaría tener que especificar cada TGTS dos veces: una en AToM^3 y otra en la herramienta de análisis. Una posibilidad sería usar AGG de manera interna. De hecho, AToM^3 permite exportar gramáticas de grafos al formato de representación que usa AGG [49]. Aun así, esta funcionalidad no es aplicable directamente sobre TGTSs ya que requiere realizar previamente un "aplanado" del grafo triple (representación de funciones de correspondencia como relaciones y de asociaciones como nodos), y construir NACs explícitas para el caso de funciones indefinidas en la parte izquierda de las reglas.

Otra opción sería desarrollar mecanismos de análisis propios para TGTSs. Aunque esto resultaría más costoso de implementar, quizás permitiría ajustar más el resultado de los análisis y acotar el rango de estudio tomando en cuenta la estructura de los grafos triples.

- Ampliar la riqueza expresiva de los patrones visuales de consulta utilizados para la obtención de vistas derivadas. Por ejemplo, algunas extensiones que pueden contribuir a ello son la generalización de restricciones positivas y negativas para actuar sobre grafos en vez de sobre elementos sencillos, la inclusión de restricciones condicionales $p \Rightarrow q$, y la utilización de conectores lógicos ($\vee$, $\wedge$) entre las restricciones de un patrón.

  Otro punto de extensión sería dotarles de la capacidad de obtener vistas derivadas que sean abstracciones de los modelos consultados, de manera que el resultado de la consulta sea una simplificación o una vista de mayor nivel de abstracción de los mismos (por ejemplo, haciendo que un conjunto de clases relacionadas se muestre como una sola clase que las representa).

- Extender el conjunto de métricas y tareas configurables en SLAMMER. Entre ellas, la inclusión de métricas subjetivas implicaría el modelado de la interacción con el usuario. También se podría definir un lenguaje imperativo para controlar el flujo de ejecución de tareas dentro de una acción, permitiendo ramificación condicional o bucles.

- Extender SLAMMER para permitir la definición de mecanismos de análisis cuyo resultado pueda disparar la ejecución de acciones, tal como se hace con los valores extremos de las métricas. Así el análisis actuaría como detector de "malos olores" en casos donde la medición es insuficiente, como por ejemplo si requiere el estudio de la consistencia dinámica de las vistas de un diseño.

  Un enfoque alternativo para el mismo propósito consistiría en extender los métodos de análisis para vistas semánticas en lenguajes multi-vista con atributos para especificar valores extremos y ejecución de acciones. La definición actual de los métodos de análisis ya permite especificar acciones que se ejecutan tras realizar el análisis, pero éstas se tienen que especificar de manera textual. No obstante, disponer de un lenguaje de acciones facilitaría la tarea, para lo cual se podría reutilizar el paquete para la definición de acciones que proporciona SLAMMER.

- Extender SLAMMER con un paquete adicional para la personalización de informes distintos de los que ya se generan. Eso permitiría obtener informes a la medida para cada tipo de métrica definido por el diseñador del entorno, configurando en cada caso la información a mostrar y el tipo de representación más adecuado de los resultados:

tabular (que es la que proporciona la implementación actual), diagramas de barras, gráficas, etc.

Un enfoque distinto para la representación de medidas sería su anotación a los modelos del sistema de un modo similar al utilizado en la herramienta MetricView [137], en cuyo caso habría que extender SLAMMER para permitir la definición de tales mecanismos de anotación.

- Mejorar la eficiencia de los entornos visuales que se generan con el prototipo implementado, principalmente en lo referente a tiempos de carga y ejecución de los sistemas de transformación de grafos triples que utilizan. Esos entornos utilizan un motor de transformación de grafos que ya existía en la herramienta y que estaba diseñado para la animación de modelos, pero que no resulta el más adecuado para el tipo de transformaciones usadas en el marco propuesto porque la mayoría se ejecutan internamente y no hay manipulación gráfica de objetos (algo que ralentiza mucho su aplicación).

  También referente a la implementación de la propuesta, una extensión que resultaría útil en campos como el modelado multi-paradigma sería ampliar el prototipo para permitir la definición de puntos de vista que no estén restringidos al meta-modelo global.

6.3.2. Líneas de investigación abiertas

Independientemente de las extensiones mencionadas en el apartado anterior sobre el trabajo ya realizado, existen varias líneas de investigación que complementarían la propuesta presentada para la generación de entornos de modelado más ricos. Como muestra se detallan algunas a continuación:

- Generación mediante meta-modelado de entornos visuales basados en patrones de diseño. Como se ha visto, el presente trabajo proponía utilizar rediseños predefinidos e integrados en los entornos generados para, por ejemplo, reestructurar automáticamente los diseños existentes de un sistema hacia patrones de diseño del dominio de aplicación. Sin embargo, en ocasiones puede resultar deseable que los entornos participen más activamente guiando al usuario en la aplicación de tales patrones *a priori* o durante el proceso de especificación de los modelos de diseño. Para ello, algunas funcionalidades interesantes en los entornos de modelado generados podrían ser:

  - facilitar un catálogo de patrones clasificados según el tipo de problema que permiten resolver y cómo aplicarlos.
  - la identificación de patrones aplicables, ya sea según se está realizando el diseño o sobre un modelo dado.

- completar automáticamente partes del diseño según patrones.

Idealmente, y siguiendo el enfoque basado en técnicas visuales utilizado en esta tesis, tales mecanismos deberían generarse a partir de una descripción de alto nivel de los patrones de diseño que complementara el meta-modelo del LVDE, al estilo de [27, 46].

- Inclusión de patrones de interacción complejos en entornos generados mediante meta-modelado. Estos entornos suelen ofrecer mecanismos de interacción fijos que dependen de la herramienta de meta-modelado utilizada para generarlos, lo cual impone una serie de restricciones en las interfaces generadas. Para ello sería necesario enriquecer la definición del LVDE con información adicional especificando, por ejemplo:

  - relaciones espaciales entre los elementos del lenguaje, como la de "contenido" en el caso de los estados de un statechart o "adyacencia" en el caso de los cuadros de un tablero de ajedrez. También se necesitaría especificar la semántica concreta de la relación dependiendo del lenguaje específico, por ejemplo para determinar si el borrado de un contenedor implica o no el borrado de todos sus contenidos [28, 29, 86].
  - paradigmas de interacción, como "objeto-acción" u "acción-objeto" [29, 86].

- Generación de entornos para lenguajes multi-vista que integren vistas gráficas (visuales) y vistas textuales. Esta necesidad viene derivada del hecho de que, al especificar un sistema, es frecuente que algunos aspectos se expresen de modo más natural utilizando una notación textual, mientras que otros sean más intuitivos mediante su representación gráfica. Por ejemplo, UML tiene partes gráficas, como los diagramas de clases o las máquinas de estados, mientras que otras partes son más adecuadas para una representación textual, como la semántica de acción o las restricciones OCL que se usan para enriquecer los diagramas.

  Así pues, el trabajo de investigación presentado en esta tesis se podría generalizar para la generación automática de entornos para LDEs multi-vista incluyendo aspectos visuales y textuales [150]. Esto implicaría extender los mecanismos proporcionados para la definición del lenguaje, la consistencia entre vistas, la propagación de cambios entre vistas del mismo o de distinto tipo, el análisis del sistema y posterior anotación de resultados ya sea sobre un modelo gráfico o sobre texto, y la definición de vistas derivadas.

Apéndice A

Introducción a Teoría de Categorías

Este apéndice incluye una introducción a diversos conceptos categóricos usados a lo largo del presente documento, principalmente en las secciones 2.2 y 4.1 y en los apéndices C y D. El apéndice comienza definiendo qué es una categoría y proporcionado diversos ejemplos; a continuación analiza los tipos más importantes de morfismo (isomorfismos, epimorfismos y monomorfismos); prosigue presentando algunas construcciones categóricas básicas, entre las que se encuentran *pushouts*, *pullbacks* y colímites; continúa con la definición de categoría HLR adhesiva; y finaliza mostrando las categorías funtor y las categorías coma como mecanismo para la construcción de nuevas categorías. Las definiciones pertenecen a [61], y los ejemplos están basados principalmente en [61, 84]. Para una descripción más detallada de estos y otros conceptos categóricos consultar [117]. Para una descripción más intuitiva e informal de los conceptos básicos consultar [123].

A.1. Categorías

Una categoría es una estructura matemática formada por objetos y morfismos, con una operación de composición sobre los morfismos y un morfismo identidad para cada objeto.

Definición 1. *(Categoría) Una* categoría $C = (Ob_C, Mor_C, \circ, id)$ *se define mediante:*

- *una clase* Ob_C *de objetos;*
- *para cada par de objetos* $A, B \in Ob_C$*, un conjunto* $Mor_C(A, B)$ *de morfismos;*
- *para todos los objetos* $A, B, C \in Ob_C$*, una operación de composición* $\circ_{(A,B,C)}$: $Mor_C(B, C) \times Mor_C(A, B) \to Mor_C(A, C)$*; y*
- *para cada objeto* $A \in Ob_C$*, un morfismo identidad* $id_A \in Mor_C(A, A)$*,*

tal que se cumplen las siguientes propiedades:

1. *asociatividad: para todos los objetos* $A, B, C, D \in Ob_C$ *y morfismos* $f\colon A \to B$*,* $g\colon B \to C$ *y* $h\colon C \to D$*, se cumple que* $(h \circ g) \circ f = h \circ (g \circ f)$*.*

2. *identidad: para todos los objetos $A, B \in Ob_C$ y morfismos $f\colon A \to B$, se cumple que $f \circ id_A = f$ y $id_B \circ f = f$.*

Observación. En vez de $f \in Mor_C(A, B)$ escribiremos $f\colon A \to B$ sin usar el índice para la composición. Para tal morfismo f, A se denomina su dominio y B su codominio.

Ejemplo. (Categorías) A continuación se muestra un conjunto de categorías de ejemplo, algunos de cuyos conceptos se usan en el apéndice C para la definición de grafos triples.

1. El ejemplo básico de categoría es la categoría **Sets**, donde los objetos son todos los conjuntos y los morfismos son las funciones $f\colon A \to B$ entre conjuntos. La composición de dos morfismos $f\colon A \to B$ y $g\colon B \to C$ se define como $(g \circ f)(x) = g(f(x))$ para todo $x \in A$. La identidad es la correspondencia idéntica $id_A\colon A \to A\colon x \mapsto x$.

2. La categoría **Graphs** tiene grafos como objetos, y morfismos de grafo como morfismos. En esta categoría, un grafo (dirigido) $G = (V, E, s, t)$ consiste en un conjunto V de nodos (también llamados vértices), un conjunto E de enlaces, y dos funciones $s, t\colon E \to V$ denominadas origen y destino:

$$E \underset{t}{\overset{s}{\rightrightarrows}} V$$

Figura A.1: *Grafo*

La figura A.2 muestra un grafo de ejemplo (usando notación gráfica) formado por el conjunto de nodos $V = \{v_1, v_2, v_3, v_4\}$, el conjunto de relaciones $E = \{e_1, e_2\}$, la función origen $s\colon E \to V\colon e_1, e_2 \mapsto v_1$, y la función destino $t\colon E \to V\colon e_1, e_2 \mapsto v_2$.

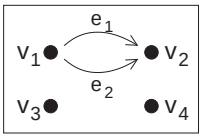


Figura A.2: *Ejemplo de grafo*

Un morfismo de grafo entre dos grafos G_1 y G_2 con $G_i = (V_i, E_i, s_i, t_i)_{i \in \{1,2\}}$ consiste en dos funciones $f_V\colon V_1 \to V_2$ y $f_E\colon E_1 \to E_2$ que preservan las funciones origen y destino, esto es, $f_V \circ s_1 = s_2 \circ f_E$ y $f_V \circ t_1 = t_2 \circ f_E$:

$$\begin{array}{ccc} E_1 & \xrightarrow{s_1} & V_1 \\ \downarrow{\scriptstyle f_E} & = & \downarrow{\scriptstyle f_V} \\ E_2 & \xrightarrow{s_2} & V_2 \end{array} \qquad \begin{array}{ccc} E_1 & \xrightarrow{t_1} & V_1 \\ \downarrow{\scriptstyle f_E} & = & \downarrow{\scriptstyle f_V} \\ E_2 & \xrightarrow{t_2} & V_2 \end{array}$$

Figura A.3: *Morfismo de grafo*

La figura A.4 muestra un morfismo de grafo. Las funciones f_V y f_E, que se muestran mediante líneas discontinuas, toman los valores $f_V \colon V_1 \to V_2 \colon v_1, v_3 \mapsto v_s; v_2, v_4 \mapsto v_t$ y $f_E \colon E_1 \to E_2 \colon e_1, e_2 \mapsto e$.

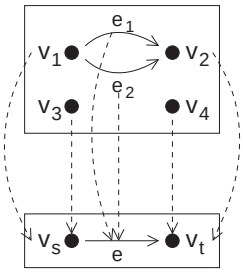


Figura A.4: *Ejemplo de morfismo de grafo*

La composición de dos morfismos de grafo $f = (f_V, f_E) \colon G_1 \to G_2$ y $g = (g_V, g_E) \colon G_2 \to G_3$ da como resultado el morfismo de grafo $g \circ f = (g_V \circ f_V, g_E \circ f_E) \colon G_1 \to G_3$. Los morfismos identidad son los correspondientes para nodos y relaciones.

3. La categoría **EGraphs** tiene E-grafos como objetos, y morfismos de E-grafo como morfismos. Un E-grafo es un grafo extendido con atributos en nodos y relaciones, y se define mediante la tupla $G = (V_G, V_D, E_G, E_{NA}, E_{EA}, (source_j, target_j)_{j \in \{G,NA,EA\}}$ (véase figura A.5) donde:

   - V_G es un conjunto de nodos de grafo
   - V_D es un conjunto de nodos de datos
   - E_G es un conjunto de relaciones de grafo
   - E_{NA} es un conjunto de relaciones de "atribución de nodos"
   - E_{EA} es un conjunto de relaciones de "atribución de relaciones"
   - $source_G \colon E_G \to V_G$, $target_G \colon E_G \to V_G$
   - $source_{NA} \colon E_{NA} \to V_G$, $target_{NA} \colon E_{NA} \to V_D$
   - $source_{EA} \colon E_{EA} \to E_G$, $target_{EA} \colon E_{EA} \to V_D$

   La figura A.6 muestra como ejemplo un E-grafo que representa la sintaxis concreta de un diagrama de secuencia (esto es, los conceptos que usa no son los del meta-modelo de UML ya que éste modela la sintaxis abstracta pero no la concreta). El E-grafo se muestra en notación gráfica, usando un tipo de flecha distinto para representar los distintos tipos de relaciones (de grafo, de atribución de nodos y de atribución de relaciones). Los nodos de datos se representan como cajas redondeadas de línea discontinua, y los nodos de grafo se representan mediante cajas.

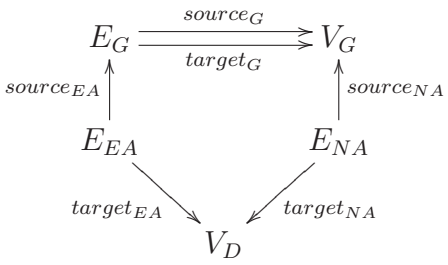


Figura A.5: *E-grafo*

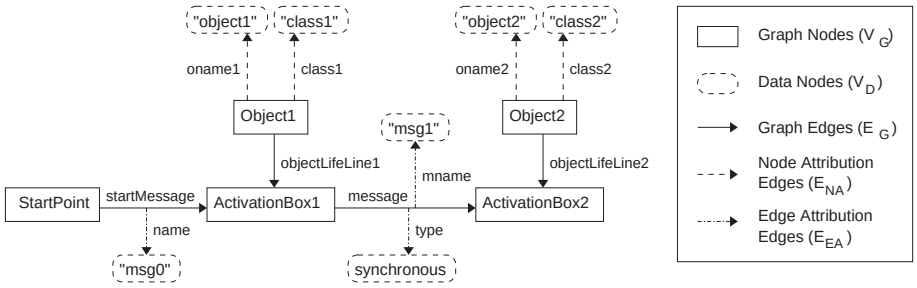


Figura A.6: *Ejemplo de E-grafo que representa un diagrama de secuencia*

Un morfismo de E-grafo entre dos E-grafos $G^i = (V_G^i, V_D^i, E_G^i, E_{NA}^i, E_{EA}^i, (source_j^i, target_j^i)_{j\in\{G,NA,EA\}})_{i\in\{1,2\}}$ es una tupla de morfismos $(f_{V_G}, f_{V_D}, f_{E_G}, f_{E_{NA}}, f_{E_{EA}})$ con $f_{V_i}: V_i^1 \to V_i^2$ y $f_{E_j}: E_j^1 \to E_j^2$ para $i \in \{G, D\}$, $j \in \{G, NA, EA\}$, donde f conmuta para todo *source* y *target* (véase figura A.7):

- $f_{V_G} \circ source_G^1 = source_G^2 \circ f_{E_G}$
- $f_{V_G} \circ target_G^1 = target_G^2 \circ f_{E_G}$
- $f_{E_G} \circ source_{EA}^1 = source_{EA}^2 \circ f_{E_{EA}}$
- $f_{V_D} \circ target_{EA}^1 = target_{EA}^2 \circ f_{E_{EA}}$
- $f_{V_G} \circ source_{NA}^1 = source_{NA}^2 \circ f_{E_{NA}}$
- $f_{V_D} \circ target_{NA}^1 = target_{NA}^2 \circ f_{E_{NA}}$

La figura A.8 muestra un ejemplo de morfismo de E-grafo. Los nodos y relaciones de los E-grafos se muestran etiquetados con un número, que es el que usa el morfismo de E-grafo para establecer las correspondencias. En este ejemplo el morfismo es no inyectivo ya que los nodos *2* y *3* del grafo tienen la misma imagen.

Existen varios mecanismos para construir nuevas categorías a partir de otras previamente definidas. Uno de esos mecanismos es la construcción denominada *categoría slice*. En ese tipo de categoría, los objetos son los morfismos de cierta categoría **C** que llegan a un objeto distinguido X, mientras que los morfismos son los morfismos de **C** que conectan objetos de **C** para obtener diagramas conmutativos.

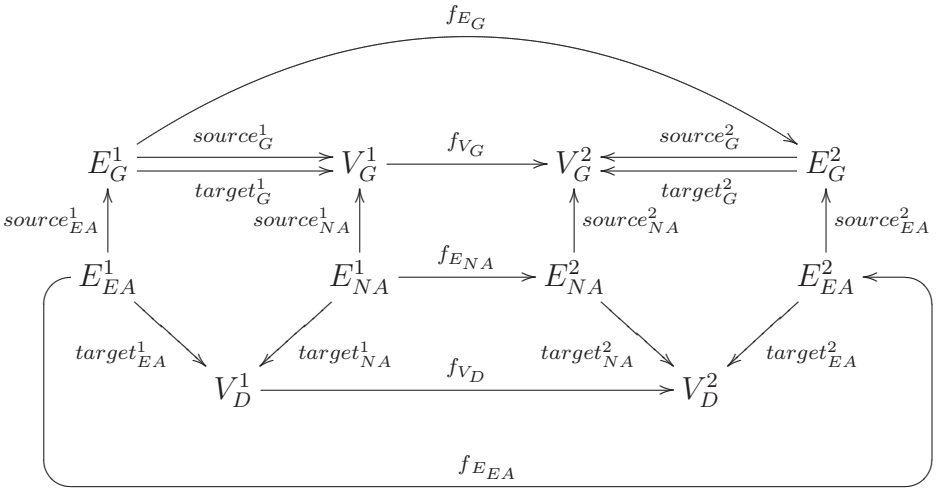


Figura A.7: *Morfismo de E-grafo*

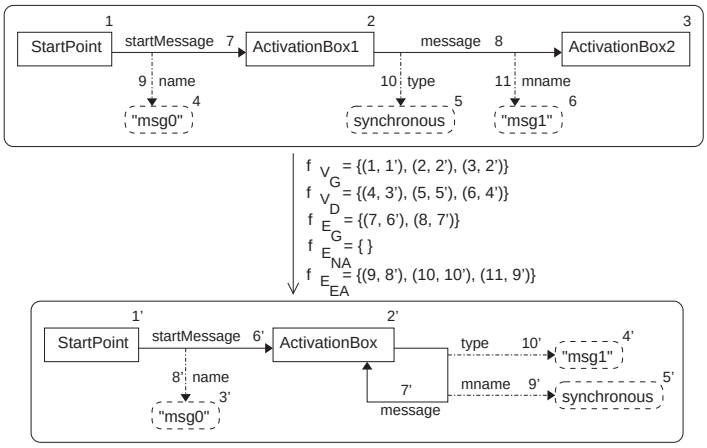


Figura A.8: *Ejemplo de morfismo de E-grafo*

Definición 2. *(Categoría slice) Dada una categoría* $\mathbf{C}$ *y un objeto* $X \in Ob_C$*, la* categoría slice $\mathbf{C}\backslash X$ *se define del siguiente modo:*

- $Ob_{C\backslash X} = \{f\colon A \to X | A \in Ob_C, f \in Mor_C(A, X)\}$;
- $Mor_{C\backslash X}(f\colon A \to X, g\colon B \to X) = \{m\colon A \to B | g \circ m = f\}$;
- *la composición de dos morfismos* $m \in Mor_{C\backslash X}(f\colon A \to X, g\colon B \to X)$ *y* $n \in Mor_{C\backslash X}(g\colon B \to X, h\colon C \to X)$ *se define como la composición en* $\mathbf{C}$ *para* $m\colon A \to B$ *y* $n\colon B \to C$*:*
- $id_{f\colon X \to A} = id_A \in Mor_C$.

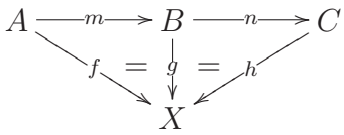


Figura A.9: *Composición de morfismos en categoría slice*

Ejemplo. (Categoría slice $\mathbf{Graphs_{TG}}$) Dada la categoría **Graphs**, podemos asignar un tipo a sus objetos eligiendo un grafo de tipos TG. Esto da lugar a la categoría $\mathbf{Graphs_{TG}}$, que se puede considerar como la categoría slice **Graphs**\TG. Cada grafo tipado se representa en esta categoría mediante su morfismo de tipado, y los morfismos de grafos de tipos son los morfismos de la categoría slice.

A.2. Monomorfismos, epimorfismos e isomorfismos

Esta sección introduce los tipos más importantes de morfismos: monomorfismos, epimorfismos e isomorfismos. Intuitivamente, dos objetos son isomorfos si tienen la misma estructura. Los morfismos que preservan esta estructura se denominan isomorfismos.

Definición 3. *(Isomorfismo) Un morfismo* $i\colon A \to B$ *se denomina* isomorfismo *si existe un morfismo* $i^{-1}\colon B \to A$ *tal que* $i \circ i^{-1} = id_B$ *e* $i^{-1} \circ i = id_A$.

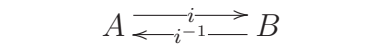


Figura A.10: *Isomorfismo*

Dos objetos A *y* B *son isomorfos, escrito* $A \cong B$*, si existe un isomorfismo* $i\colon A \to B$.

Observación. Si i es un isomorfismo, entonces también es un monomorfismo y un epimorfismo. Para cada isomorfismo i, el isomorfismo inverso i^{-1} es único.

Ejemplo. (Isomorfismos) Los isomorfismos en **Sets**, **Graphs** y $\mathbf{Graphs_{TG}}$ son los morfismos que son inyectivos y sobreyectivos (para cada componente de la categoría). Los isomorfismos en las categorías slice son los isomorfismos de las categorías subyacentes.

Definición 4. *(Monomorfismo) Dada una categoría* $\mathbf{C}$*, un morfismo* $m\colon B \to C$ *se denomina* monomorfismo *si, para todos los morfismos* $f, g\colon A \to B \in Mor_C$*, se cumple que* $m \circ f = m \circ g \Rightarrow f = g$*:*

Figura A.11: *Monomorfismo*

Definición 5. *(Epimorfismo) Dada una categoría* **C**, *un morfismo* $e\colon A \to B$ *se denomina* epimorfismo *si, para todos los morfismos* $f, g\colon B \to C \in Mor_C$, *se cumple que* $f \circ e = g \circ e \Rightarrow f = g$*:*

$$A \xrightarrow{e} B \underset{g}{\overset{f}{\rightrightarrows}} C$$

Figura A.12: *Epimorfismo*

Ejemplo. (Monomorfismos e isomorfismos)

- En la categoría **Sets**, los monomorfismos son todas las funciones inyectivas, y los epimorfismos son todas las funciones sobreyectivas. La figura A.13 muestra sendos ejemplos de monomorfismo y epimorfismo en esta categoría.

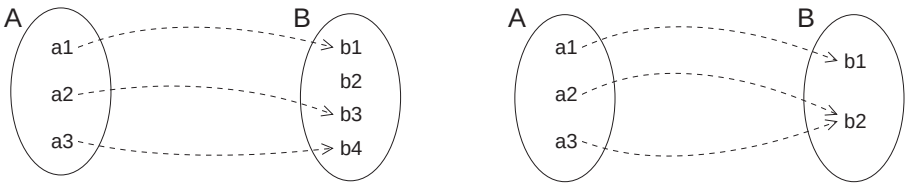


Figura A.13: *Ejemplo de monomorfismo (izquierda) y epimorfismo (derecha) en* **Sets**

- En las categorías **Graphs** y $\mathbf{Graphs_{TG}}$, los monomorfismos son todos los morfismos inyectivos (esto es, inyectivos en los conjuntos V y E), y los epimorfismos son todos los morfismos sobreyectivos (esto es, sobreyectivos en los conjuntos V o E). La figura A.14 muestra sendos ejemplos de monomorfismo y epimorfismo en la categoría **Graphs**.

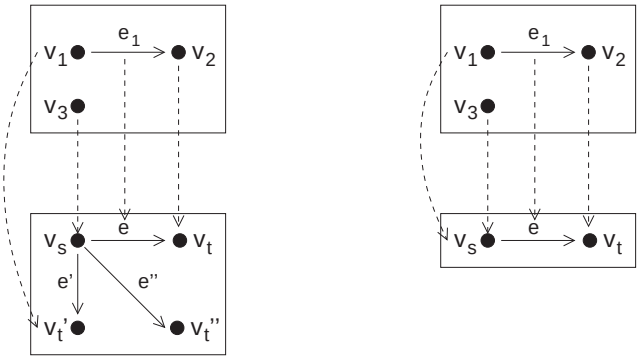


Figura A.14: *Ejemplo de monomorfismo (izquierda) y epimorfismo (derecha) en* **Graphs**

- En una categoría slice, los monomorfismos son los monomorfismos de la categoría subyacente. Los epimorfismos de la categoría subyacente son epimorfismos en la categoría slice.

A.3. Algunas construcciones categóricas

Para poder aplicar una regla de transformación de grafos a un grafo se necesita una técnica que permita unir dos grafos a través de un subgrafo común. El concepto categórico *pushout* generaliza la operación de unión de dos objetos. Intuitivamente, un pushout es el objeto resultante de unir dos objetos a través de un subobjeto común. Un pullback es su construcción dual.

Definición 6. *(Pushout) Dados dos morfismos* $f\colon A \to B$ *y* $g\colon A \to C \in Mor_C$*, un* pushout (D, f', g') *sobre* f *y* g *se define mediante:*

- *un objeto pushout* D *y*
- *morfismos* $f'\colon C \to D$ *y* $g'\colon B \to D$ *con* $f' \circ g = g' \circ f$*,*

tal que la siguiente propiedad universal se cumple: para todos los objetos X *con morfismos* $h\colon B \to X$ *y* $k\colon C \to X$ *con* $k \circ g = h \circ f$*, existe un único morfismo* $x\colon D \to X$ *tal que* $x \circ g' = h$ *y* $x \circ f' = k$*:*

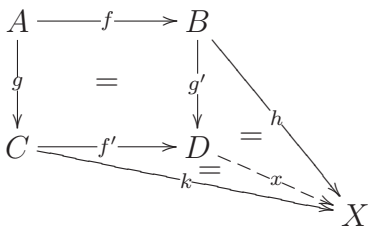


Figura A.15: *Pushout*

Observación. El objeto pushout D es único salvo isomorfismos. Eso significa que si (X, k, h) también es un pushout sobre f y g, entonces $x\colon D \to X$ es un isomorfismo con $x \circ g' = h$ y $x \circ f' = k$. Viceversa, si (D, f', g') es un pushout sobre f y g y $x\colon D \to X$ es un isomorfismo, entonces (X, k, h) también es un pushout sobre f y g, donde $h = x \circ g'$ y $k = x \circ f'$.

Ejemplo. (Construcciones pushout)

- En la categoría **Sets**, un pushout sobre los morfismos $f\colon A \to B$ y $g\colon A \to C$ se construye como sigue. Sea

$$\sim_{f,g} = t(\{(a_1, a_2) \in A \times A | f(a_1) = f(a_2) \vee g(a_1) = g(a_2)\})$$

  el cierre transitivo de $Kern(f)$ y $Kern(g)$; $\sim_{f,g}$ es una relación de equivalencia. Entonces, el objeto D y los morfismos se definen como:

- $D = A|_{\sim_{f,g}} \dot{\bigcup} B \backslash f(A) \dot{\bigcup} C \backslash g(A)$, donde $\dot{\bigcup}$ es la unión disjunta;
- $f' \colon C \to D \colon x \mapsto \begin{cases} [a] & : \exists a \in A : g(a) = x \\ x & : \textit{en otro caso} \end{cases}$;
- $g' \colon B \to D \colon x \mapsto \begin{cases} [a] & : \exists a \in A : f(a) = x \\ x & : \textit{en otro caso} \end{cases}$.

La figura A.16 muestra un ejemplo de construcción pushout en **Sets** para los conjuntos $A = \{a1, a2, a3\}$, $B = \{b1, b2\}$ y $C = \{c1, c2, c3, c4\}$, y los morfismos $f \colon A \to B$ con $f(a1) = b1$, $f(a2) = f(a3) = b2$, y $g \colon A \to C$ con $g(a1) = c3$, $g(a2) = c2$, $g(a3) = c1$. Para construir el pushout seguimos los siguientes pasos:

1. Definimos la relación $\sim \forall a \in A$: $b1 \sim c3$, $b2 \sim c2$, $b2 \sim c1$.
2. Definimos la relación de equivalencia generada por $\sim$: $b1 \equiv c3$, $b2 \equiv c2 \equiv c1$; por tanto $[b1] = [c3] = (b1, c3)$, $[b2] = [c1] = [c2] = (b2, c1, c2)$,$[c4] = (c4)$.
3. Construimos el objeto pushout: $PO = \{(b1, c3), (b2, c1, c2), (c4)\}$.

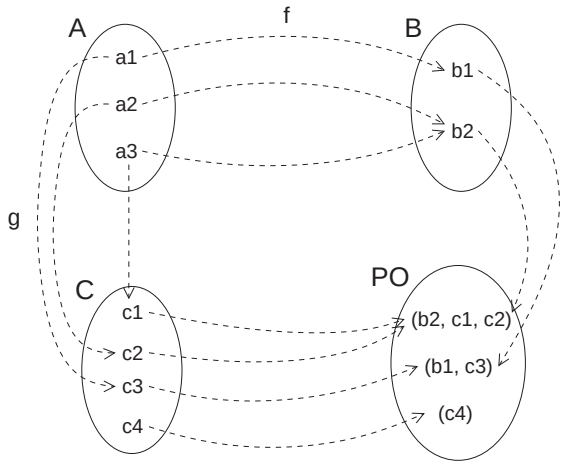


Figura A.16: *Ejemplo de pushout en la categoría* **Sets**

- En las categorías **Graphs** y $\mathbf{Graphs_{TG}}$, un pushout se construye componente a componente en **Sets** para cada uno de sus conjuntos de nodos y relaciones.

  La figura A.17 muestra un ejemplo de construcción pushout en la categoría **Graphs**. Para facilitar la legibilidad, la figura sólo incluye los morfismos que corresponden a nodos del grafo, pero no a relaciones.

- Si la categoría **C** tiene pushouts, los pushouts en la categoría slice $\mathbf{C}\backslash X$ pueden construirse a partir de los pushouts en **C**. Dados los objetos $f \colon A \to X$, $g \colon B \to X$ y $h \colon C \to X$, y los morfismos m y n en $\mathbf{C}\backslash X$ como muestra (1) en la figura A.18, se

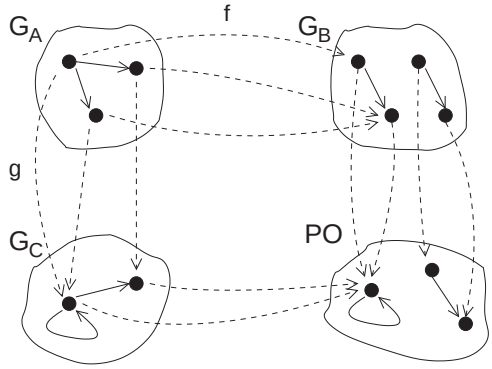


Figura A.17: *Ejemplo de pushout en la categoría* **Graphs**

cumple que $g \circ m = f = h \circ n$ por la definición de morfismos en $\mathbf{C}\backslash X$. Primero se construye el pushout (2) de la figura A.18 en $\mathbf{C}$ sobre $C \xleftarrow{n} A \xrightarrow{m} B$. A partir de (2) se obtiene el morfismo inducido $d\colon D \to X$ como el objeto pushout, y los morfismos s y t con $d \circ s = g$ y $d \circ t = h$, lo que lleva al pushout (1) en $\mathbf{C}\backslash X$:

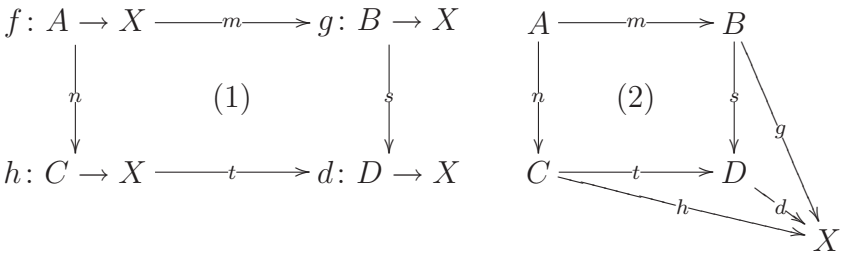


Figura A.18: *Construcción pushout en categoría slice*

La construcción inversa de un pushout se denomina complemento pushout, y se utiliza en DPO para modelar el borrado de elementos.

Definición 7. *(Complemento pushout) Dados dos morfismos $f\colon A \to B$ y $n\colon B \to D$, su* complemento pushout *es $A \xrightarrow{m} C \xrightarrow{g} D$ si (1) en la figura A.19 es un pushout:*

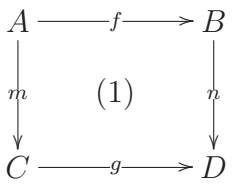


Figura A.19: *Complemento pushout*

La construcción dual de un pushout se denomina pullback. Intuitivamente, un pullback es la intersección generalizada de objetos a través de un objeto común.

Definición 8. *(Pullback) Dados dos morfismos $f\colon C \to D$ y $g\colon B \to D$, un* pullback *(A, f', g') sobre f y g se define mediante:*

- *un objeto pullback A y*
- *morfismos $f'\colon A \to B$ y $g'\colon A \to C$ con $g \circ f' = f \circ g'$,*

tal que la siguiente propiedad universal se cumple: para todos los objetos X con morfismos $h\colon X \to B$ y $k\colon X \to C$ con $f \circ k = g \circ h$, existe un único morfismo $x\colon X \to A$ tal que $f' \circ x = h$ y $g' \circ x = k$:

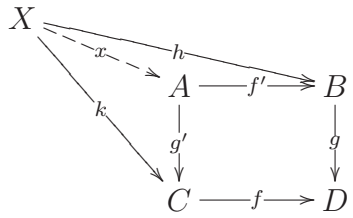


Figura A.20: *Pullback*

Ejemplo. (Construcciones pullback)

- En la categoría **Sets**, el pullback $C \xleftarrow{\pi_g} A \xrightarrow{\pi_f} B$ sobre los morfismos $f\colon C \to D$ y $g\colon B \to D$ se construye mediante $A = \bigcup_{d \in D} f^{-1}(d) \times g^{-1}(d)$ con morfismos $f'\colon A \to B\colon (x, y) \mapsto y$ y $g'\colon A \to C\colon (x, y) \mapsto x$.

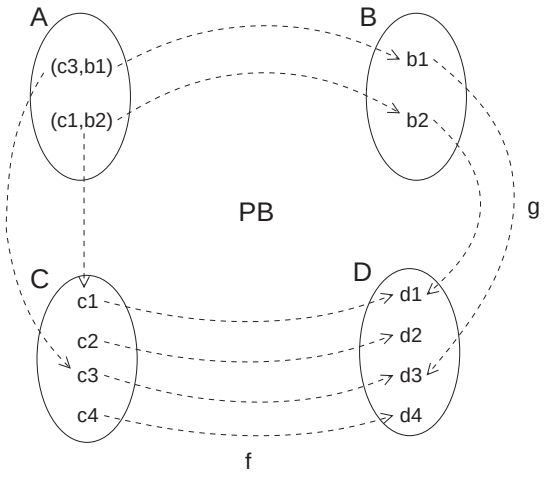


Figura A.21: *Ejemplo de pullback en la categoría* **Sets**

La figura A.21 muestra un ejemplo de construcción de un pullback en **Sets** para los conjuntos $B = \{b1, b2\}$, $C = \{c1, c2, c3, c4\}$ y $D = \{d1, d2, d3, d4\}$, y los morfismos $f\colon C \to D$ con $f(c1) = d1$, $f(c2) = d2$, $f(c3) = d3$, $f(c4) = d4$ y $g\colon B \to D$ con $g(b1) = d3$, $g(b2) = d1$.

- En las categorías **Graphs** y $\mathbf{Graphs_{TG}}$, un pullback se construye componente a componente en **Sets** para cada uno de sus conjuntos de nodos y relaciones.

  La figura A.22 muestra un ejemplo de construcción pullback en la categoría **Graphs**. Para facilitar la legibilidad, la figura sólo incluye los morfismos que corresponden a nodos del grafo, pero no a relaciones.

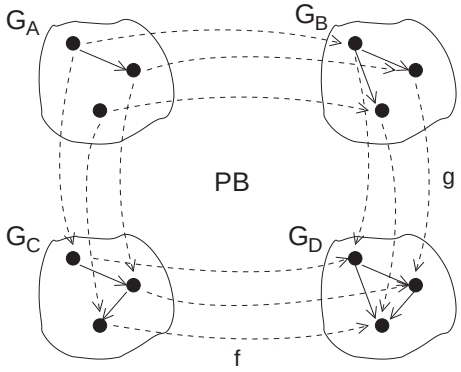


Figura A.22: *Ejemplo de pullback en la categoría* **Graphs**

- En una categoría slice, la construcción de pullbacks es dual a la construcción de pushouts si la categoría subyacente tiene pullbacks.

Definición 9. *(Cocono) Un* cocono $(X, \{g_i\})$ *de un diagrama con objetos* A_i *y morfismos* f_k *es un objeto* X *y una familia de morfismos* $g_i \colon A_i \to X$ *coherentes con* f_k*, esto es,* $g_i = g_j \circ f_k$ *para cada* $f_k \colon A_i \to A_j$*, tal como muestra la figura A.23.*

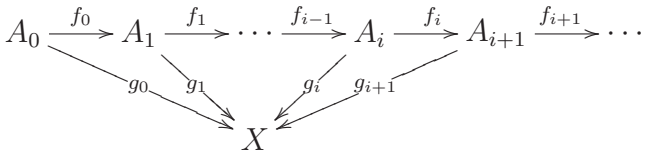


Figura A.23: *Cocono*

Definición 10. *(Colímite) El* colímite *de un diagrama es un cocono inicial a través del cual cualquier otro cocono factoriza. Esto es, un colímite es un cocono* $(X, \{g_i\})$ *tal que para cualquier otro cocono* $(Y, \{h_i\})$ *existe un único morfismo* $\alpha \colon X \to Y$ *tal que* $\alpha \circ g_i = h_i$ *para todo objeto* A_i*, tal como muestra la figura A.24.*

A.4. Categorías HLR adhesivas

En [61, 62, 110] la teoría general de transformación de grafos se generalizó para reescribir no sólo grafos, sino objetos de cualquier categoría HLR adhesiva (por ejemplo grafos

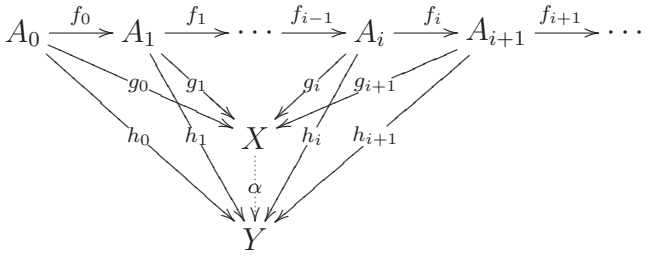


Figura A.24: *Colímite*

tipados atribuidos, hipergrafos, redes de Petri, etc.). Esta sección recoge su definición. Posteriormente, la sección C.3 demostrará que grafos triples y morfismos de grafo triple forman una categoría HLR adhesiva, y por tanto sus objetos se pueden manipular utilizado los resultados obtenidos para la transformación de grafos.

La idea intuitiva de categoría adhesiva es la de una categoría con pushouts y pullbacks compatibles los unos con los otros. Su definición se basa en los cuadrados de van Kampen. Un cuadrado van Kampen es una estructura categórica con un pushout estable bajo pullbacks, y viceversa, pullbacks estables bajo pushouts y pullbacks combinados.

Definición 11. *(Cuadrado van Kampen) Un pushout (1) es un* cuadrado van Kampen *si, para cualquier cubo conmutativo (2) con (1) en la parte inferior y donde las caras traseras son pullbacks, se cumple que la cara superior es un pushout si y sólo si las caras frontales son pullbacks (véase figura A.25).*

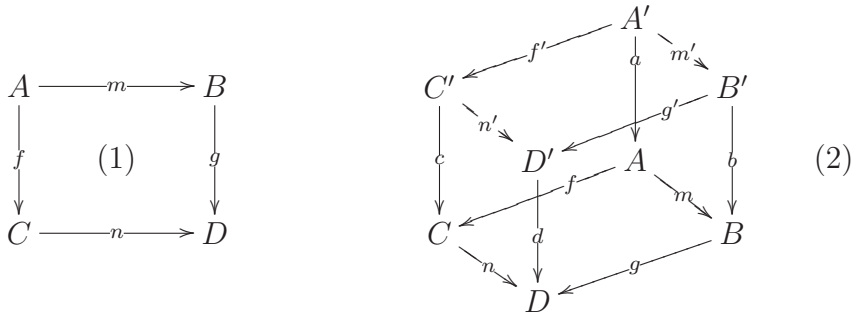


Figura A.25: *Cuadrado van Kampen*

Definición 12. *(Categoría adhesiva) Una categoría* **C** *es una* categoría adhesiva *si:*

1. **C** *tiene pushouts sobre monomorfismos (esto es, pushouts donde al menos uno de los morfismos es un monomorfismo).*

2. **C** *tiene pullbacks.*

3. *Los pushouts sobre monomorfismos son cuadrados van Kampen.*

La principal diferencia entre las categorías adhesivas y las categorías adhesivas de reemplazo de alto nivel (HLR) es que estas últimas consideran una clase distinguida $\mathcal{M}$ de monomorfismos en vez de todos los monomorfismos, de tal modo que sólo tienen que ser cuadrados van Kampen los pushouts sobre $\mathcal{M}$-morfismos. Además, sólo se necesitan pullbacks sobre $\mathcal{M}$-morfismos, y no sobre morfismos arbitrarios.

Definición 13. *(Categoría HLR adhesiva) Una categoría* $\mathbf{C}$ *con una clase de morfismos* $\mathcal{M}$ *se llama* categoría HLR adhesiva *si:*

1. $\mathcal{M}$ *es una clase de monomorfismos cerrada bajo isomorfismos, composición* ($f\colon A \to B \in \mathcal{M}, g\colon B \to C \in \mathcal{M} \Rightarrow g \circ f \in \mathcal{M}$*), y descomposición* ($g \circ f \in \mathcal{M}, g \in \mathcal{M} \Rightarrow f \in \mathcal{M}$*).*

2. $\mathbf{C}$ *tiene pushouts y pullbacks sobre* $\mathcal{M}$*-morfismos, y los* $\mathcal{M}$*-morfismos son cerrados bajo pushouts y pullbacks.*

3. *Los pushouts en* $\mathbf{C}$ *sobre* $\mathcal{M}$*-morfismos son cuadrados van Kampen.*

Observación. Un pushout *sobre* un $\mathcal{M}$-morfismo es un pushout donde al menos uno de los morfismos dados está en $\mathcal{M}$. Los pushouts son cerrados bajo $\mathcal{M}$-morfismos si, dado un pushout, $m \in \mathcal{M}$ implica que $n \in \mathcal{M}$ (véase parte izquierda de la figura A.25). Análogamente, los pullbacks son cerrados bajo $\mathcal{M}$-morfismos si, dado un pullback, $n \in \mathcal{M}$ implica que $m \in \mathcal{M}$.

Ejemplo. (Categorías HLR adhesivas) **Sets**, **Graphs** y $\mathbf{Graphs_{TG}}$ son categorías HLR adhesivas (véase [61]).

Definición 14. *(Categoría HLR adhesiva débil) Una categoría* $\mathbf{C}$ *con una clase de morfismos* $\mathcal{M}$ *se denomina* categoría HLR adhesiva débil *si:*

- $\mathcal{M}$ *es una clase de monomorfismos cerrada bajo isomorfismos, composición* ($f\colon A \to B \in \mathcal{M}, g\colon B \to C \in \mathcal{M} \Rightarrow g \circ f \in \mathcal{M}$*), y descomposición* ($g \circ f \in \mathcal{M}, g \in \mathcal{M} \Rightarrow f \in \mathcal{M}$*).*

- $\mathbf{C}$ *tiene pushouts y pullbacks sobre* $\mathcal{M}$*-morfismos, y los* $\mathcal{M}$*-morfismos son cerrados bajo pushouts y pullbacks.*

- *Los pushouts en* $\mathbf{C}$ *sobre* $\mathcal{M}$*-morfismos son cuadrados van Kampen débiles, esto es, la propiedad del cuadrado van Kampen se cumple para todos los cubos conmutativos con* $m \in \mathcal{M}$ *y* ($f \in \mathcal{M}$ *o* $b, c, d \in \mathcal{M}$*) (véase definición 11).*

Por definición, todas las categorías HLR adhesivas también son categorías HLR adhesivas débiles.

A.5. Funtores, categorías funtor y categorías coma

Las categorías coma son un mecanismo para la construcción de nuevas categorías. En este tipo de categorías, los objetos son un tipo de morfismo llamado funtor que relaciona dos categorías. Esta sección presenta las categorías coma y otros conceptos relacionados, ya que en la sección C.3 se utilizan para demostrar que grafos triples y morfismos de grafo triple son una categoría HLR adhesiva.

Un funtor es una función entre dos categorías compatible con la composición y las identidades.

Definición 15. *(Funtor) Sean dos categorías* $\mathbf{C}$ *y* $\mathbf{D}$*. Un* funtor $F\colon \mathbf{C} \to \mathbf{D}$ *se define mediante la dupla* $F = (F_{Ob}, F_{Mor})$ *con:*

- *una función* $F_{Ob}\colon Ob_C \to Ob_D$ *y*
- *una función* $F_{Mor(A,B)}\colon Mor_C(A,B) \to Mor_D(F_{Ob}(A), F_{Ob}(B))$ *de los morfismos para cada par de objetos* $A, B \in Ob_C$*,*

tales que:

1. *Para todos los morfismos* $f\colon A \to B$ *y* $g\colon B \to C \in Mor_C$ *se cumple que* $F(g \circ f) = F(g) \circ F(f)$*.*
2. *Para todos los objetos* $A \in Ob_C$ *se cumple que* $F(id_A) = id_{F(A)}$*.*

Observación. Por simplicidad, escribiremos $F(A)$ y $F(B)$ sin poner los índices para objetos y morfismos.

Las transformaciones naturales se utilizan para comparar funtores. Funtores y transformaciones naturales forman la categoría funtor.

Definición 16. *(Transformación natural) Sean dos categorías* $\mathbf{C}$ *y* $\mathbf{D}$ *y dos funtores* $F, G\colon \mathbf{C} \to \mathbf{D}$*. Una* transformación natural $\alpha\colon F \Rightarrow G$ *es una familia de morfismos* $\alpha = (\alpha_A)_{A \in Ob_C}$ *con* $\alpha_A\colon F(A) \to G(A) \in Mor_D$*, tal que para todos los morfismos* $f\colon A \to B \in Mor_C$ *se cumple que* $\alpha_B \circ F(f) = G(f) \circ \alpha_A$*:*

$$\begin{array}{ccc} F(A) & \xrightarrow{F(f)} & F(B) \\ \downarrow{\scriptstyle \alpha_A} & & \downarrow{\scriptstyle \alpha_B} \\ G(A) & \xrightarrow{G(f)} & G(B) \end{array}$$

Figura A.26: *Condición para transformaciones naturales*

Definición 17. *(Categoría funtor) Sean dos categorías* $\mathbf{C}$ *y* $\mathbf{D}$*. La categoría funtor* $[\mathbf{C}, \mathbf{D}]$ *tiene la clase de todos los funtores* $F\colon \mathbf{C} \to \mathbf{D}$ *como objetos, y las transformaciones naturales como morfismos. La composición de dos transformaciones naturales* $\alpha\colon F \Rightarrow G$ *y* $\beta\colon G \Rightarrow H$ *se calcula componiendo cada componente en* $\mathbf{D}$*, lo cual significa que* $\beta \circ \alpha = (\beta_A \circ \alpha_A)_{A \in Ob_C}$*. Las identidades están dadas por las transformaciones naturales definidas para cada componente sobre las identidades* $id_{F(A)} \in \mathbf{D}$*.*

Definición 18. *(Categoría coma) Sean dos funtores* $F\colon \mathbf{A} \to \mathbf{C}$ *y* $G\colon \mathbf{B} \to \mathbf{C}$ *y un conjunto índice I. La categoría coma* $\mathbf{ComCat}(\mathbf{F}, \mathbf{G}; I)$ *tiene la clase de todas las tripletas* (A, B, op) *como objetos, con* $A \in Ob_A$*,* $B \in Ob_B$ *y* $op = [op_i]_{i \in I}$ *para* $op_i \in Mor_C(F(A), G(B))$*; un morfismo* $f\colon (A, B, op) \to (A', b', op')$ *en* $\mathbf{ComCat}(\mathbf{F}, \mathbf{G}; I)$ *es un par* $f = (f_A\colon A \to A', f_B\colon B \to B')$ *de morfismos en* $\mathbf{A}$ *y* $\mathbf{B}$ *tales que* $G(f_B) \circ op_i = op'_i \circ F(f_A)$ *para todo* $i \in I$ *(véase figura A.27).*

$$\begin{array}{ccc} F(A) & \xrightarrow{op_i} & F(B) \\ \Big\downarrow{\scriptstyle F(f_A)} & = & \Big\downarrow{\scriptstyle G(f_B)} \\ G(A) & \xrightarrow{op'_i} & G(B) \end{array}$$

Figura A.27: *Condición para morfismos en una categoría coma*

La composición de morfismos en $\mathbf{ComCat}(\mathbf{F}, \mathbf{G}; I)$ *se define para cada componente, y las identidades son pares de identidades en las categorías* $\mathbf{A}$ *y* $\mathbf{B}$*.*

Apéndice B

Introducción a Signaturas y Álgebras

En este apéndice se incluye una breve introducción a signaturas algebraicas y álgebras, ya que éstas se utilizan en el apéndice C para definir el concepto de grafo triple atribuido (básico para la transformación de grafos triples). El apéndice recoge únicamente los conceptos necesarios para tal definición. Las definiciones y ejemplos están sacados de [61].

B.1. Signaturas algebraicas

Una signatura es la descripción sintáctica de un álgebra o, por otro lado, la descripción formal de la interfaz de un programa. Consiste en un conjunto de símbolos y operaciones.

Definición 19. *(Signatura algebraica) Una* signatura algebraica *o* signatura $\sum = (S, OP)$ *consiste en un conjunto S de tipos y una familia $OP = (OP_{w,s} \in S^* \times S)$ de símbolos de operaciones.*

Observación. Dado un símbolo de operación $op \in OP_{w,s}$ escribimos $op\colon w \to s$. Si $w = \lambda$ entonces $op\colon \to s$ se denomina símbolo constante.

Ejemplo.

1. La signatura de los números naturales tiene un tipo denominado *nat* y la constante *cero*. Las operaciones disponibles son sucesor, suma y multiplicación.

```
NAT =
sorts : nat
opns :
cero: → nat
suc: nat → nat
suma: nat nat → nat
mult: nat nat → nat
```

2. La signatura de los caracteres tiene un tipo denominado *char*, la constante *a*, y una operación *siguiente* que obtiene el carácter siguiente a uno dado.

```
CHAR =
sorts : char
opns :
```

```
a: → char
sig: char → char
```

3. La signatura de las cadenas de caracteres utiliza la signatura de los caracteres anterior. Esto significa que todos los tipos y operaciones definidas sobre caracteres pueden usarse en la nueva signatura. El tipo de la signatura para cadenas de caracteres es *string*, con la constante cadena vacía, y las operaciones de concatenación, inserción de caracteres en una cadena y obtención del primer carácter de una cadena.

```
STRING =
sorts : string
opns :
vacía: → string
concat: string string → string
inserta: char string → string
primero: string → char
```

Definición 20. *(Morfismo de signatura) Sean las signaturas* $\sum = (S, OP)$ *y* $\sum' = (S', OP')$. *Un* morfismo de signatura $h\colon \sum \to \sum'$ *es una tupla* $h = (h_S\colon S \to S', h_{OP}\colon OP \to OP')$ *tal que* $h_{OP}(f)\colon h_S(s_1)...h_S(s_n) \to h_S(s) \in OP'$ *para todo* $f\colon s_1...s_n \to s \in OP$.

Ejemplo. Sean las signaturas NAT y $STRING$ del ejemplo anterior. Definimos el morfismo de signatura $h = (h_S, h_{OP})\colon STRING \to NAT$, con $h_S(char) = h_S(string) = nat$, $h_{OP}(a) = h_{OP}(vacia) = cero$, $h_{OP}(sig) = h_{OP}(primero) = suc$ y $h_{OP}(concat) = h_{OP}(inserta) = suma$.

Definición 21. *(Categoría* **Sig***) Signaturas y morfismos de signatura forman la categoría* **Sig** *de signaturas.*

B.2. Álgebras

Un álgebra es un modelo semántico de una signatura o, en otras palabras, implementa una signatura.

Definición 22. *(*$\sum$*-álgebra) Dada una signatura* $\sum = (S, OP)$, *una* $\sum$-álgebra $A = ((A_s)_{s \in S}, (op_A)_{op \in OP})$ *se define mediante:*

- *por cada tipo* $s \in S$, *un conjunto* A_s *llamado conjunto portador;*
- *por cada símbolo constante* $c\colon \to s \in OP$, *una constante* $c_A \in A_s$;
- *por cada símbolo de operación* $op\colon s_1...s_n \to s \in OP$, *una función* $op_A\colon A_{s_1} \times ... \times A_{s_n} \to A_s$.

Distintas álgebras pueden implementar una misma signatura dotándola de distintas semánticas. Para analizar las relaciones entre álgebras se definen los homomorfismos.

Definición 23. *(Homomorfismo) Sean una signatura $\sum = (S, OP)$ y dos $\sum$-álgebras A y B. Un homomorfismo $h\colon A \to B$ es una familia $h = (h_s)_{s \in S}$ de funciones $h_s\colon A_s \to B_s$ tal que las siguientes propiedades se cumplen:*

- *para cada símbolo constante $c\colon \to s \in OP$, $h_s(c_A) = c_B$;*
- *para cada símbolo de operación $op\colon s_1...s_n \to s \in OP$, se cumple que $h_s(op_A(x_1, ..., x_n)) = op_B(h_{s_1}(x_1), ..., h_{s_n}(x_n))$ para todo $x_i \in A_{s_i}$.*

Definición 24. *(Categoría $\mathbf{Alg}(\sum)$) Dada una signatura $\sum$, $\sum$-álgebras y homomorfismos forman la categoría $\mathbf{Alg}(\sum)$.*

Ejemplo. Como ejemplo, a continuación se muestran sendas álgebras para las signaturas $CHAR$ y $STRING$.

1. Una posible implementación de la signatura $CHAR$ es el siguiente álgebra C:
   $\mathtt{C}_{char}$ = {a, ..., z, A, ..., Z, 0, 1, ... 9}
   $\mathtt{a}_C$ = A $\in$ $\mathtt{C}_{char}$
   $\mathtt{sig}_C$: $\mathtt{C}_{char} \to \mathtt{C}_{char}$; a $\mapsto$ b, ..., z $\mapsto$ A, A $\mapsto$ B, ..., Y $\mapsto$ Z, Z $\mapsto$ 0, 0 $\mapsto$ 1, ..., 9 $\mapsto$ a

2. El $STRING$-álgebra D se define sobre caracteres, al igual que el álgebra C:
   $\mathtt{D}_{char}$ = {a, ..., z, A, ..., Z, 0, 1, ... 9}
   $\mathtt{D}_{string}$ = $\mathtt{D}^*_{char}$
   $\mathtt{a}_D$ = A $\in$ $\mathtt{D}_{char}$
   $\mathtt{vacia}_D$ = $\lambda \in$ $\mathtt{D}_{string}$
   $\mathtt{sig}_D$: $\mathtt{D}_{char} \to \mathtt{D}_{char}$; a $\mapsto$ b, ..., z $\mapsto$ A, A $\mapsto$ B, ..., Y $\mapsto$ Z, Z $\mapsto$ 0, 0 $\mapsto$ 1, ..., 9 $\mapsto$ a
   $\mathtt{concat}_D$: $\mathtt{D}_{string} \times \mathtt{D}_{string} \to \mathtt{D}_{string}$; (s,t) $\mapsto$ st
   $\mathtt{inserta}_D$: $\mathtt{D}_{char} \times \mathtt{D}_{string} \to \mathtt{D}_{string}$; (x,s) $\mapsto$ xs
   $\mathtt{primero}_D$: $\mathtt{D}_{string} \to \mathtt{D}_{string}$; $\lambda \mapsto$ A, s $\mapsto$ $\mathtt{s}_1$ con s = $\mathtt{s}_1 \ldots \mathtt{s}_n$

A continuación se define el concepto de álgebra final, un tipo especial de álgebra que usa el apéndice C para la atribución de grafos triples.

Definición 25. *(Álgebra final) Dada una signatura $\sum = (S, OP)$, el $\sum$-álgebra final Z se define mediante:*

- *$Z_s = \{s\}$ para cada tipo $s \in S$;*
- *$c_Z = s \in Z_s$ para un símbolo constante $c\colon \to s \in OP$;*

- $op_Z : \{s_1\} \times ... \times \{s_n\}$: ($s_1...s_n \mapsto s$ *para cada símbolo de operación* op: $s_1...s_n \rightarrow s \in OP$.

Ejemplo. A continuación se muestra el álgebra final para la signatura $STRING$.

$\mathtt{Z}_{char}$ = {char}
$\mathtt{Z}_{string}$ = {string}
$\mathtt{a}_Z$ = char $\in \mathtt{Z}_{char}$
$\mathtt{vacia}_Z$ = string $\in \mathtt{Z}_{string}$
$\mathtt{sig}_Z : \mathtt{Z}_{char} \rightarrow \mathtt{Z}_{char}$; char $\mapsto$ char
$\mathtt{concat}_Z : \mathtt{Z}_{string} \times \mathtt{Z}_{string} \rightarrow \mathtt{Z}_{string}$; (string,string) $\mapsto$ string
$\mathtt{inserta}_Z : \mathtt{Z}_{char} \times \mathtt{Z}_{string} \rightarrow \mathtt{Z}_{string}$; (char,string) $\mapsto$ string
$\mathtt{primero}_Z : \mathtt{Z}_{string} \rightarrow \mathtt{Z}_{char}$; string $\mapsto$ char

Apéndice C

Transformación de grafos triples tipados atribuidos

Este apéndice recoge la formalización y extensión de las gramáticas de grafos triples [162] siguiendo el enfoque algebraico *Double Pushout*. La formalización se ha realizado dentro de la presente tesis como base teórica del marco para la definición de LVDEs multi-vista presentado en la sección 4.1.

Las gramáticas de grafos triples [162] se crearon para modelar la evolución sincronizada de dos grafos etiquetados atribuidos. Para ello se utilizaba un tercer grafo, denominado grafo correspondencia, cuyos nodos tenían morfismos a los nodos de los otros dos grafos. Además, las reglas de una gramática de este tipo estaban restringidas a ser monotónicas, esto es, podían borrar pero no eliminar elementos. En este apéndice extendemos esa definición a sistemas de transformación de grafos triples con las siguientes características añadidas:

1. especificación de morfismos desde los nodos del grafo correspondencia no sólo a nodos, sino también a aristas de los otros dos grafos, así como la posibilidad de dejar morfismos indefinidos. La razón de esto es que, en primer lugar, el marco para la formalización de lenguajes multi-vista requiere ser capaz de relacionar enlaces en un grafo con nodos y/o aristas en otro grafo. Por ejemplo, supongamos que existen dos aristas atribuidas con los mismos nodos origen y destino en una vista del sistema. En ese caso no basta con relacionar los nodos origen y destino de la arista que están en la vista con los del repositorio, ya que si se modifica el atributo de una arista en la vista se necesita identificar unívocamente a cuál de las dos aristas del repositorio propagar los cambios. Por tanto, relacionar aristas con aristas en el marco multi-vista propuesto es crucial.

   Por otro lado, los usuarios pueden borrar elementos de las vistas que también existen en el repositorio (y con los cuales están relacionados mediante un grafo correspondencia). Al borrar un elemento de una vista la función de correspondencia a la vista queda indefinida, pero la función al repositorio sigue estando definida. Esto es útil porque permite identificar posteriormente qué elemento se ha borrado y realizar el procesamiento adecuado en cada caso (por ejemplo decrementar el contador del elemento en

el repositorio para un borrado conservativo). Además, saber que un morfismo está indefinido es una potente restricción negativa implícita en la parte izquierda de una regla para la que no hace falta definir una NAC.

2. asignación de un tipo a los grafos triples, similar al concepto de meta-modelo en paradigmas de meta-modelado. El apéndice D mostrará una extensión al concepto de meta-modelo presentado aquí, que incluye relaciones de herencia entre nodos y aristas. Esto permite que los elementos de una regla se apliquen a los elementos de un grafo que tienen el mismo tipo o cualquiera de sus subtipos concretos, obteniendo así una notación compacta para un conjunto de reglas similares.

3. formalización de las reglas de grafos triples utilizando el enfoque algebraico *Double Pushout*. Esto hace posible reutilizar muchos de los resultados clásicos de la teoría de transformación de grafos sobre grafos triples.

4. extensión de las reglas triples con condiciones de aplicación, al estilo de las definidas en [93].

5. extensión de la definición de gramática de grafo triple para permitir reglas que no sean monotónicas, y que puedan usarse para transformaciones de modelos genéricas (en vez de sólo para la especificación de reglas operacionales).

El apéndice comienza definiendo el concepto de grafo triple y los morfismos de grafos triples basándose en la definición de E-grafo, el cual permite tener atributos en nodos y relaciones. Grafos triples y morfismos de grafos triples forman la categoría $\mathbf{TriAGraphs_{TriATG}}$. Después se muestra cómo construir *pushouts* y *pullbacks* en esta categoría, necesarios para definir un sistema de transformación de grafos triples. A continuación se demuestra que la categoría $\mathbf{TriAGraphs_{TriATG}}$ es una categoría de reemplazo de alto nivel (HLR) adhesiva, demostrando para ello que grafos triples y morfismos de grafos triples son isomorfos a una categoría coma. Esta demostración permite usar muchos de los resultados obtenidos para la transformación de grafos sobre la formalización de grafos triples realizada, ya que en [61] se generalizan para categorías HLR adhesivas. De manera ilustrativa se presentarán explícitamente algunos de estos resultados para grafos triples. Las diversas secciones se ilustran con ejemplos basados en los diagramas de secuencia UML. El lector puede consultar en el apéndice A las definiciones de los conceptos categóricos básicos utilizados a lo largo de este apéndice.

C.1. Grafos triples tipados atribuidos

En esta sección se formaliza el concepto de grafo triple tipado atribuido utilizando teoría de categorías. Para ello primero se define el objeto TriE-grafo, formado por tres E-grafos

y dos funciones de correspondencia c_1 y c_2. Las funciones de correspondencia se definen desde los nodos de uno de los E-grafos (denominado grafo correspondencia) a los nodos y relaciones de los otros dos E-grafos. Las funciones también pueden estar indefinidas, lo que se modela con un elemento especial en el codominio representado mediante "·".

Definición 26. *(TriE-grafo) Un* TriE-grafo $TriG = (G_1, G_2, G_C, c_1, c_2)$ *está formado por tres E-grafos* $G_i = (V_{G_i}, V_{D_i}, E_{G_i}, E_{NA_i}, E_{EA_i}, (source_{j_i}, target_{j_i})_{j\in\{G,NA,EA\}})$ *para* $i \in \{1, 2, C\}$*, con* $V_{D_1} = V_{D_2} = V_{D_C}$*, y dos funciones* $c_j \colon V_{G_C} \to V_{G_j} \cup E_{G_j} \cup \{\cdot\}$ *(para* $j = 1, 2$*).*

El grafo G_1 *se denomina origen,* G_2 *se denomina destino, y* G_C *se denomina correspondencia. Las funciones* c_1 *y* c_2 *se llaman funciones de correspondencia origen y destino, respectivamente. Definimos los conjuntos auxiliares* $edges_i = \{x \in V_{G_C} | c_i(x) \in E_{G_i}\}$*,* $nodes_i = \{x \in V_{G_C} | c_i(x) \in V_{G_i}\}$ *y* $undef_i = \{x \in V_{G_C} | c_i(x) = \cdot\}$ *para* $i = 1, 2$*. Contienen el conjunto de nodos* x *del grafo correspondencia cuya imagen para la función* c_i *es un nodo, una relación o está indefinido, respectivamente.*

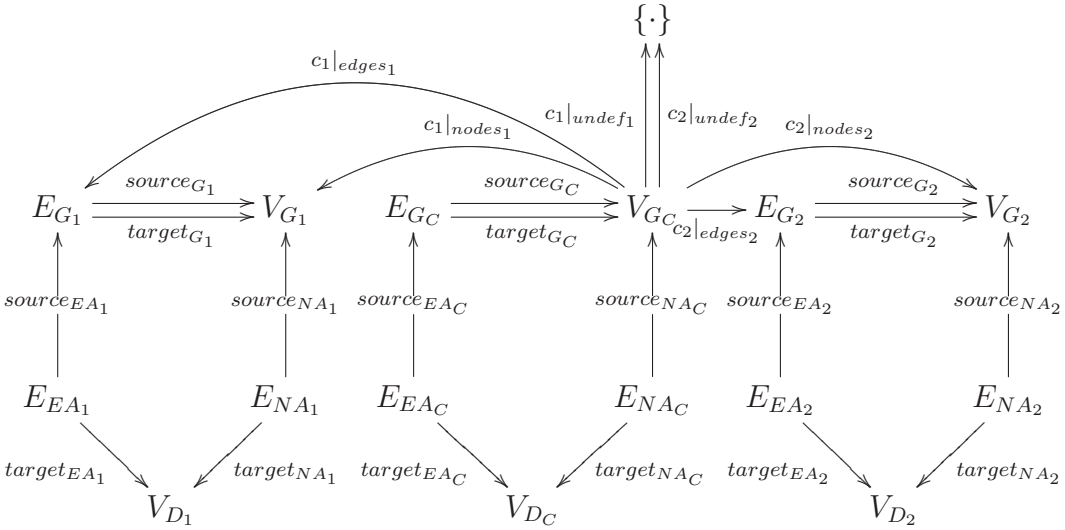


Figura C.1: *TriE-grafo*

Los morfismos c_1 y c_2 representan relaciones m-a-n entre los nodos y relaciones de G_1 y G_2 a través de G_C del siguiente modo: $x \in V_{G_1} \cup E_{G_1}$ está relacionado con $y \in V_{G_2} \cup E_{G_2} \iff \exists z \in V_{G_C} \mid x = c_1(z) \wedge y = c_2(z)$. Asumimos que todos los conjuntos de datos V_{D_i} son el mismo, pero los diferenciamos en tres conjuntos separados para poder reutilizar los conceptos desarrollados en [62] sobre E-grafos.

La figura C.2 muestra un TriE-grafo (usando una representación gráfica) que define la sintaxis abstracta y concreta de un diagrama de secuencia UML. El grafo origen G_1 en la parte inferior corresponde a la sintaxis concreta, el grafo destino G_2 en la parte superior corresponde a la sintaxis abstracta, y el grafo correspondencia G_C en la parte

intermedia contiene nodos que relacionan los elementos de G_1 y G_2 mediante funciones de correspondencia. Por razones de claridad, aunque $V_{D_1} = V_{D_2} = V_{D_C}$, sus elementos se representan en todos los E-grafos donde se utilizan. Sin embargo los elementos de los tres conjuntos son los mismos, así que por ejemplo el nodo de datos "$class1''$ en V_{D_1} es el mismo nodo que "$class1''$ en V_{D_2}. Además, pese a que los conjuntos V_{D_i} pueden ser infinitos, sólo se representan los elementos que se usan.

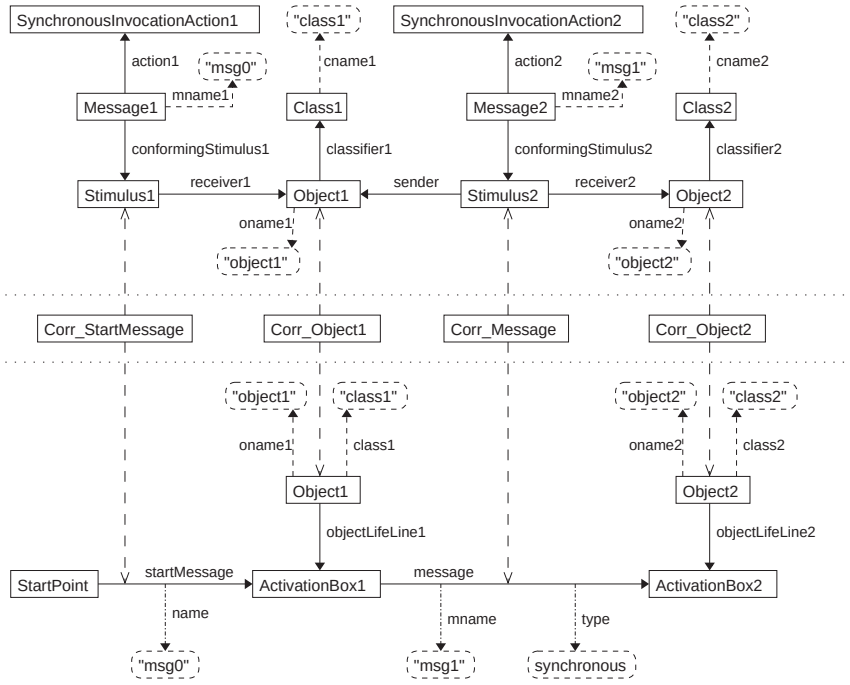


Figura C.2: *Ejemplo de TriE-grafo*

A continuación se definen los morfismos de TriE-grafo, compuestos por tres morfismos de E-grafo más un conjunto de restricciones adicionales que preservan las funciones de correspondencia.

Definición 27. *(Morfismo de TriE-grafo) Sean dos TriE-grafos* $TriG^i = (G_1^i, G_2^i, G_C^i, c_1^i, c_2^i)$ *con* $i = 1, 2$. *Un* morfismo de TriE-grafo $f\colon TriG^1 \to TriG^2$ *se define mediante la tupla* $f = (f^1, f^2, f^c)$ *formada por tres morfismos de E-grafo* $f^i\colon G_i^1 \to G_i^2$ $(i \in \{1, 2, C\})$ *tales que:*

- $f^i_{V_{G_i}} \circ c_i^1|_{nodes_i^1} = c_i^2 \circ f^C_{V_{G_C}}|_{nodes_i^1}$ *para* $i = 1, 2$.
- $f^i_{E_{G_i}} \circ c_i^1|_{edges_i^1} = c_i^2 \circ f^C_{V_{G_C}}|_{edges_i^1}$ *para* $i = 1, 2$.
- $c_i^1|_{undef_i^1} = c_i^2 \circ f^C_{V_{G_C}}|_{undef_i^1}$ *para* $i = 1, 2$.

tal como muestra la figura C.3.

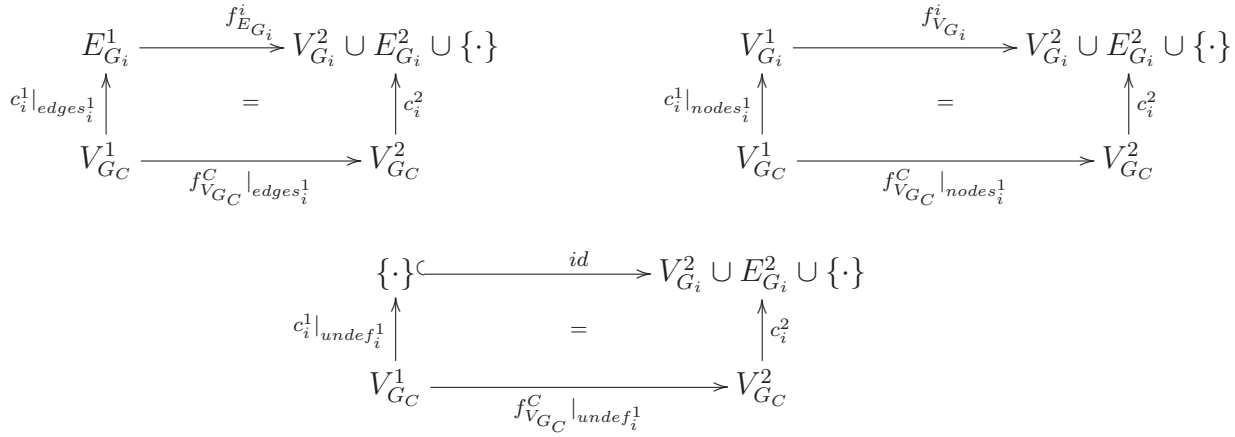


Figura C.3: *Condiciones que debe cumplir un morfismo de TriE-grafo*

Observación. f^1, f^2 y f^C son morfismos de E-grafo, y por tanto añaden restricciones adicionales (no mostradas en la figura C.3) que preservan la estructura de los tres E-grafos que forman un TriE-grafo.

TriE-grafos y morfismos de TriE-grafo forman una categoría, donde los primeros son los objetos y los segundos los morfismos. Además, la composición de morfismos de TriE-grafo es asociativa y existe un morfismo identidad para cada objeto TriE-grafo.

Definición 28. *(Categoría* **TriEGraphs***) TriE-grafos y morfismos de TriE-grafo forman la categoría* **TriEGraphs***.*

Demostración. La categoría **TriEGraphs** está formada por la clase de todos los TriE-grafos y la clase $\biguplus_{(A,B)\in TriEGraphs\times TriEGraphs}[A,B]_{TriEGraphs}$ de todos los morfismos de TriE-grafo, donde $[A,B]_{TriEGraphs}$ es el conjunto de morfismos de TriE-grafo que van de A a B. Para demostrar que *TriEGraphs* es una categoría hay que comprobar que: (i) los morfismos de TriE-grafo se pueden componer, (ii) son asociativos, y (iii) existen morfismos identidad.

(i) Composición de morfismos de TriE-grafo. Sean $f\colon TriG^1 \to TriG^2$ y $g\colon TriG^2 \to TriG^3$ dos morfismos de TriE-grafo. La composición $g \circ f = (g^1 \circ f^1, g^2 \circ f^2, g^c \circ f^c)\colon TriG^1 \to TriG^3$ se define como la composición de los tres morfismos de E-grafo en f y g. Como muestran las siguientes tres fórmulas, el morfismo h resultante es consistente con la definición 27 de morfismo de TriE-grafo:

- $(g^i_{V_{G_i}} \circ f^i_{V_{G_i}}) \circ c^1_i|_{nodes^1_i} = c^3_i \circ g^C_{V_{G_C}}|_{nodes^2_i} \circ f^C_{V_{G_C}}|_{nodes^1_i} = c^3_i \circ (g^C_{V_{G_C}} \circ f^C_{V_{G_C}})|_{nodes^1_i}$ para $i = 1, 2$ (véase figura C.4). La igualdad $g^C_{V_{G_C}}|_{nodes^2_i} \circ f^C_{V_{G_C}}|_{nodes^1_i} = (g^C_{V_{G_C}} \circ f^C_{V_{G_C}})|_{nodes^1_i}$ se cumple porque los morfismos de TriE-grafo requieren que $f^C_{V_{G_C}}|_{nodes^1_i}(V^1_{G_C})$ sea un subconjunto de $nodes^2_i$ (primera condición de la definición 27). La conmutatividad del cuadrado exterior se deriva de la conmutatividad de los dos cuadrados interiores.

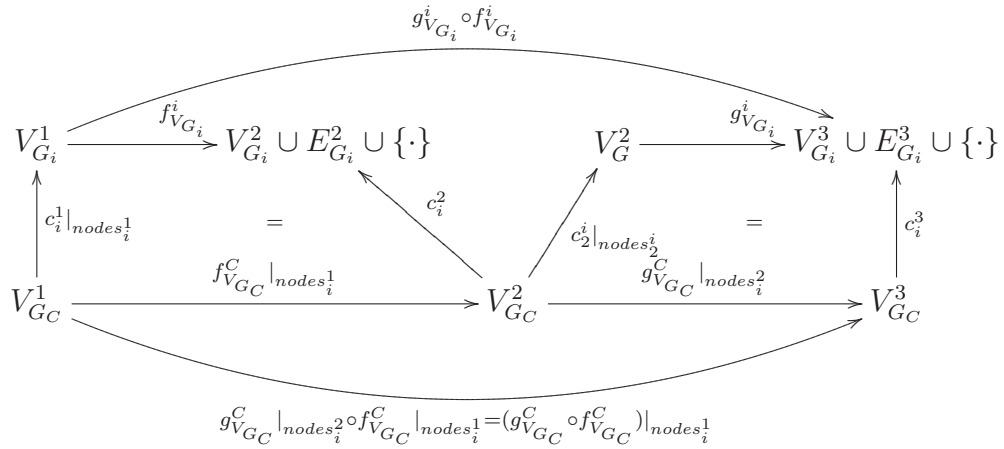


Figura C.4: *Composición de morfismos de TriE-grafo: (a) condición para nodos de correspondencia cuya imagen es un nodo*

- $(g^i_{E_{G_i}} \circ f^i_{E_{G_i}}) \circ c^1_i|_{edges^1_i} = c^3_i \circ g^C_{V_{G_C}}|_{edges^2_i} \circ f^C_{V_{G_C}}|_{edges^1_i} = c^3_i \circ (g^C_{V_{G_C}} \circ f^C_{V_{G_C}})|_{edges^1_i}$ para $i = 1, 2$ (véase figura C.5). La igualdad $g^C_{V_{G_C}}|_{edges^2_i} \circ f^C_{V_{G_C}}|_{edges^1_i} = (g^C_{V_{G_C}} \circ f^C_{V_{G_C}})|_{edges^1_i}$ se cumple porque los morfismos de TriE-grafo requieren que $f^C_{V_{G_C}}|_{edges^1_i}(V^1_{G_C})$ sea un subconjunto de $edges^2_i$ (segunda condición de la definición 27). La conmutatividad del cuadrado exterior se deriva de la conmutatividad de los dos cuadrados interiores.

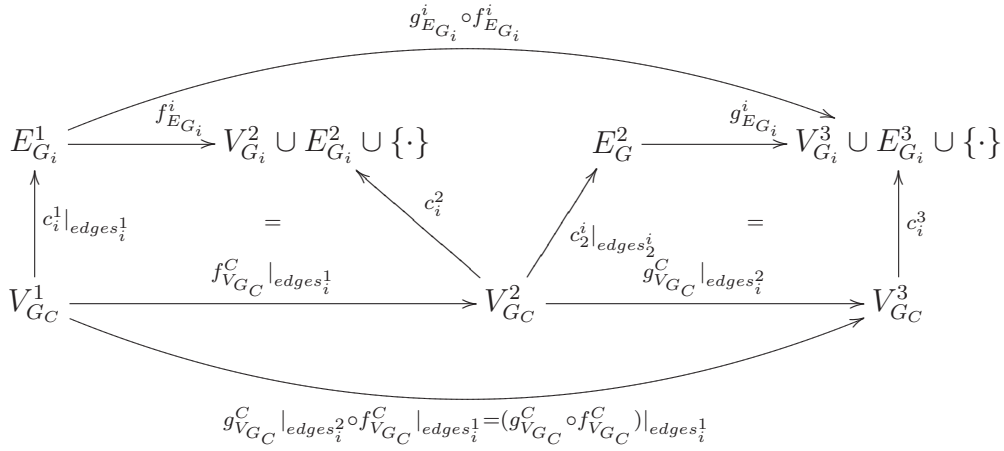


Figura C.5: *Composición de morfismos de TriE-grafo: (b) condición para nodos de correspondencia cuya imagen es una relación*

- $c^1_i|_{undef^1_i} = c^3_i \circ (g^C_{V_{G_C}}|_{undef^2_i} \circ f^C_{V_{G_C}}|_{undef^1_i}) = c^3_i \circ (g^C_{V_{G_C}} \circ f^C_{V_{G_C}})|_{undef^1_i}$ para $i = 1, 2$ (véase figura C.6). La igualdad $g^C_{V_{G_C}}|_{undef^2_i} \circ f^C_{V_{G_C}}|_{undef^1_i} = (g^C_{V_{G_C}} \circ f^C_{V_{G_C}})|_{undef^1_i}$ se cumple porque los morfismos de TriE-grafo requieren que $f^C_{V_{G_C}}|_{undef^1_i}(V^1_{G_C})$ sea un subconjunto de $undef^2_i$ (tercera condición de la definición 27). La conmutatividad del cuadrado exterior se deriva de la conmutatividad de los dos cuadrados interiores.

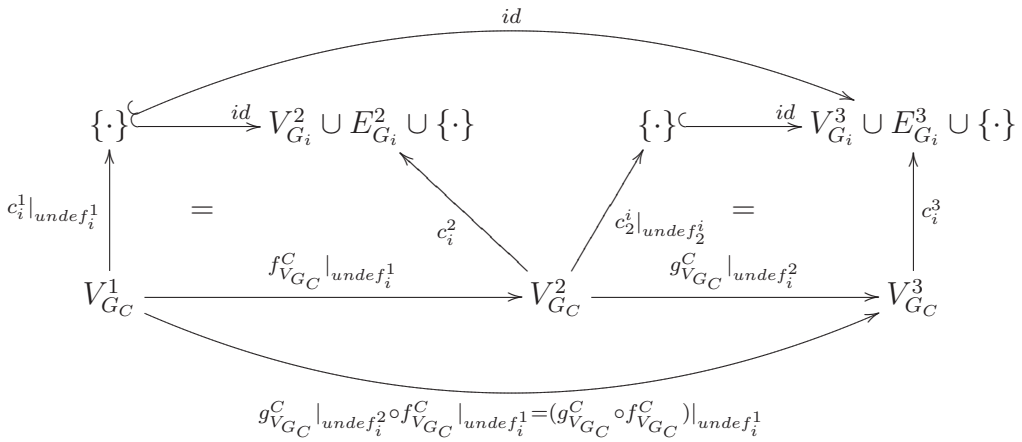


Figura C.6: *Composición de morfismos de TriE-grafo: (c) condición para nodos de correspondencia cuya imagen está indefinida*

(ii) Asociatividad de morfismos de TriE-grafo. Sean tres morfismos de TriE-grafo $f = (f^1, f^2, f^C)\colon G^1 \to G^2$, $g = (g^1, g^2, g^C)\colon G^2 \to G^3$ y $h = (h^1, h^2, h^C)\colon G^3 \to G^4$, hay que demostrar que $(h \circ g) \circ f = h \circ (g \circ f)$. Esto se deriva de la asociatividad de los tres morfismos de E-grafo que forman un morfismo de TriE-grafo. La figura C.7 muestra el caso en que los nodos de correspondencia tienen como imagen un nodo. En la figura puede verse cómo la asociatividad de los morfismos de TriE-grafo se reduce a la asociatividad de los morfismos de E-grafo f^i y f^C que lo forman, los cuales son asociativos porque E-grafos y morfismos de E-grafo forman una categoría. El caso para los nodos de correspondencia cuya imagen es una relación o está indefinida es similar al que muestra la figura C.7.

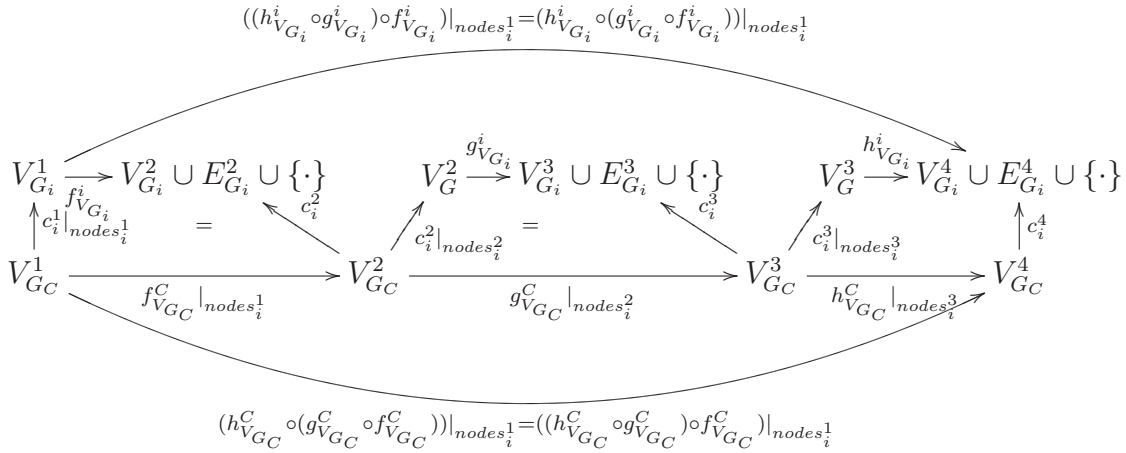


Figura C.7: *Asociatividad de morfismos de TriE-grafo: (a) condición para nodos de correspondencia cuya imagen es un nodo*

De manera formal se tiene que:

- $((h^i_{V_{G_i}} \circ g^i_{V_{G_i}}) \circ f^i_{V_{G_i}}) \circ c^1_i|_{nodes^1_i} = (h^i_{V_{G_i}} \circ (g^i_{V_{G_i}} \circ f^i_{V_{G_i}})) \circ c^1_i|_{nodes^1_i} =$

$c_i^4 \circ ((h^C_{V_{G_C}} \circ (g^C_{V_{G_C}} \circ f^C_{V_{G_C}}))|_{nodes_i^1}) = c_i^4 \circ (((h^C_{V_{G_C}} \circ g^C_{V_{G_C}}) \circ f^C_{V_{G_C}})|_{nodes_i^1})$, ya que cada morfismo sencillo es un morfismo de E-grafo, y éstos son asociativos.

- $((h^i_{E_{G_i}} \circ g^i_{E_{G_i}}) \circ f^i_{E_{G_i}}) \circ c_i^1|_{edges_i^1} = (h^i_{E_{G_i}} \circ (g^i_{E_{G_i}} \circ f^i_{E_{G_i}})) \circ c_i^1|_{edges_i^1} =$
  $c_i^4 \circ ((h^C_{V_{G_C}} \circ (g^C_{V_{G_C}} \circ f^C_{V_{G_C}}))|_{edges_i^1}) = c_i^4 \circ (((h^C_{V_{G_C}} \circ g^C_{V_{G_C}}) \circ f^C_{V_{G_C}})|_{edges_i^1})$, ya que cada morfismo sencillo es un morfismo de E-grafo, y éstos son asociativos.

- $c_i^1|_{undef_i^1} = c_i^4 \circ (((h^C_{V_{G_C}} \circ g^C_{V_{G_C}}) \circ f^C_{V_{G_C}})|_{undef_i^1}) = c_i^4 \circ ((h^C_{V_{G_C}} \circ (g^C_{V_{G_C}} \circ f^C_{V_{G_C}}))|_{undef_i^1})$, ya que cada morfismo sencillo es un morfismo de E-grafo, y éstos son asociativos.

Por tanto $(h \circ g) \circ f = h \circ (g \circ f)$.

(iii) Identidades en **TriEGraphs**. Hay que demostrar que, para cada TriE-grafo G, existe el morfismo identidad $id_G \colon G \to G$ tal que para cualesquiera TriE-grafos G, H y morfismos $f \colon G \to H$, se cumple que $f \circ id_G = id_H \circ f = f$.

Dado un TriE-grafo G, el morfismo identidad $id_G = (id^1_G, id^2_G, id^C_G)$ se construye tomando los morfismos identidad de cada E-grafo que forma el grafo triple. Dado un morfismo de TriE-grafo arbitrario $f = (f^1, f^2, f^C) \colon G \to H$ se obtiene que:

- Al usar identidades de E-grafos, $f^i \circ id^i_G = f^i$ para $i \in \{1, 2, C\}$. Por tanto $(f^1, f^2, f^C) \circ (id^1_G, id^2_G, id^C_G) = (f^1 \circ id^1_G, f^2 \circ id^2_G, f^C \circ id^C_G) = (f^1, f^2, f^C)$, y así $f \circ id_G = f$.

- Al usar identidades de E-grafos de nuevo, $id^i_H \circ f^i = f^i$ para $i \in \{1, 2, C\}$. Por tanto $(id^1_H, id^2_H, id^C_H) \circ (f^1, f^2, f^C) = (id^1_H \circ f^1, id^2_H \circ f^2, id^C_H \circ f^C) = (f^1, f^2, f^C)$, y así $id_H \circ f = f$.

□

A continuación se dota a los TriE-grafos de un álgebra para dar estructura al conjunto de atributos y proporcionar operaciones apropiadas para su cómputo. Se asigna un solo álgebra para todos los E-grafos del TriE-grafo, de tal modo que cada conjunto de atributos V_{D_i} contiene la unión de los conjuntos portadores del álgebra.

Definición 29. *(Grafo triple atribuido) Sea una signatura de datos $DSIG = (S_D, OP_D)$ con tipos de valores de atributos $S'_D \subseteq S_D$. Un* grafo triple atribuido *$TriAG = (TriG, D)$ es un TriE-grafo $TriG = (G_1, G_2, G_C, c_1, c_2)$ y un álgebra D de las signaturas $DSIG$ dadas, con $\biguplus_{s \in S'_D} D_s = V_{D_i}$ para $i \in \{1, 2, C\}$.*

Por la definición de TriE-grafo tenemos que $V_{D_1} = V_{D_2} = V_{D_C}$. Además $AG_i = (G_i, D)$ para $i \in \{1, 2, C\}$ son tres grafos atribuidos. Aunque podrían usarse tres álgebras distintas para cada E-grafo del TriE-grafo, se usa sólo una para simplificar la teoría.

A continuación se definen los morfismos de grafo triple atribuido, compuestos por un morfismo de TriE-grafo y un homomorfismo de álgebra.

Definición 30. *(Morfismo de grafo triple atribuido) Sean dos grafos triples atribuidos* $TriAG^i = (TriG^i, D^i)$ *con* $i = 1, 2$. *Un* morfismo de grafo triple atribuido $f\colon TriAG^1 \to TriAG^2$ *es una tupla* $f = (f_{TriG}, f_D)$ *donde* $f_{TriG}\colon TriG^1 \to TriG^2$ *es un morfismo de TriE-grafo y* $f_D\colon D^1 \to D^2$ *es un homomorfismo de álgebra tal que el diagrama de la figura C.8 conmuta para todo* $s \in S'_D$.

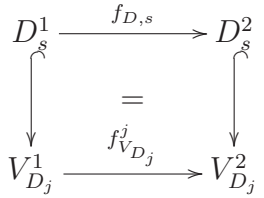


Figura C.8: *Condición para morfismos de grafo triple atribuido*

Grafos triples atribuidos y morfismos de grafo triple atribuido forman una categoría, donde los primeros son los objetos y los segundos los morfismos. Además, la composición de morfismos de grafo triple atribuido es asociativa, y existe un morfismo identidad para cada objeto grafo triple atribuido.

Definición 31. *(Categoría* **TriAGraphs***) Grafos triples atribuidos y morfismos de grafo triple atribuido forman la categoría* **TriAGraphs**.

Demostración. La categoría **TriAGraphs** está formada por la clase de todos los grafos triples atribuidos y la clase $\biguplus_{(A,B)\in TriAGraphs\times TriAGraphs}[A, B]_{TriAGraphs}$ de todos los morfismos de grafo triple atribuido, donde $[A, B]_{TriAGraphs}$ es el conjunto de morfismos de grafo triple atribuido que van de A a B. Para demostrar que *TriAGraphs* es una categoría hay que comprobar que: (i) los morfismos de **TriAGraphs** se pueden componer, (ii) son asociativos, y (iii) existen morfismos identidad.

(i) Composición de morfismos de **TriAGraphs**. Sean dos morfismos de grafo triple atribuido $f\colon TriAG^1 \to TriAG^2$ y $g\colon TriAG^2 \to TriAG^3$. La composición $g \circ f = (g_{TriG} \circ f_{TriG} = (g^1 \circ f^1, g^2 \circ f^2, g^c \circ f^c), g_D \circ f_D)\colon TriAG^1 \to TriAG^3$ se define como la composición de los correspondientes morfismo de TriE-grafo y homomorfismo de álgebra en f y g. $g_{TriG} \circ f_{TriG}$ es un morfismo de TriE-grafo ya que este tipo de morfismos es cerrado bajo composición. Por otro lado, $g_D \circ f_D\colon D^1 \to D^3$ es de nuevo un homomorfismo de álgebra que cumple el diagrama de la figura C.9, donde la conmutatividad de los dos cuadrados asegura la conmutatividad del cuadrado exterior. Por tanto $g \circ f$ es un morfismo de grafo triple atribuido.

(ii) Asociatividad de morfismos de **TriAGraphs**. Dados tres morfismos de grafo triple atribuido $f = (f_{TriG} = (f^1, f^2, f^C), f_D)\colon G^1 \to G^2$, $g = (g_{TriG} = (g^1, g^2, g^C), g_D)\colon G^2 \to G^3$ y $h = (h_{TriG} = (h^1, h^2, h^C), h_D)\colon G^3 \to G^4$, hay que demostrar que $(h \circ g) \circ f = h \circ (g \circ f)$.

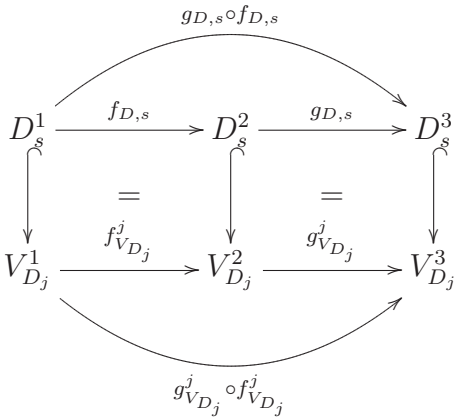


Figura C.9: *Composición de morfismos de grafo triple atribuido: condición para los homomorfismos de álgebra*

Esto se deriva de la asociatividad del morfismo de TriE-grafo y del homomorfismo de álgebra que lo forman. De manera formal se tiene que:
$(h \circ g) \circ f = ((h_{TriG}, h_D) \circ (g_{TriG}, g_D)) \circ (f_{TriG}, f_D) =$
$(h_{TriG} \circ g_{TriG}, h_D \circ g_D) \circ (f_{TriG}, f_D) =$
$(h_{TriG} \circ g_{TriG} \circ f_{TriG}, h_D \circ g_D \circ f_D) =$
$(h_{TriG} \circ (g_{TriG} \circ f_{TriG}), h_D \circ (g_D \circ f_D)) =$
$(h_{TriG}, h_D) \circ (g_{TriG} \circ f_{TriG}, g_D \circ f_D) =$
$(h_{TriG}, h_D) \circ ((g_{TriG}, g_D) \circ (f_{TriG}, f_D)) =$
$h \circ (g \circ f)$.

Por tanto $(h \circ g) \circ f = h \circ (g \circ f)$.

(iii) Identidades en **TriAGraphs**. Hay que demostrar que, para cada grafo triple atribuido G, existe el morfismo identidad $id_G \colon G \to G$ tal que para cualesquiera grafos triples atribuidos G, H y morfismos $f \colon G \to H$, se cumple que $f \circ id_G = id_H \circ f = f$.

Dado un grafo triple atribuido G, el morfismo $id_G = (id_G^{TriG} = (id_G^1, id_G^2, id_G^C), id_D)$ se construye tomando el morfismo identidad de su TriE-grafo y el homomorfismo identidad de su álgebra. Dado un morfismo de grafo triple atribuido arbitrario $f = (f_{TriG} = (f^1, f^2, f^3), f_D) \colon G \to H$ se obtiene que:

- $id_G \circ f = (id_G^{TriG}, id_D) \circ (f_{TriG}, f_D) = (id_G^{TriG} \circ f_{TriG}, id_D \circ f_D) = (f_{TriG}, f_D) = f$.
- $f \circ id_H = (f_{TriG}, f_D) \circ (id_H^{TriG}, id_H) = (f_{TriG} \circ id_H^{TriG}, f_D \circ id_H) = (f_{TriG}, f_D) = f$.

□

A continuación se dota a los grafos triples atribuidos de un tipo. Esto se modela mediante un grafo triple atribuido distinguido llamado grafo triple de tipos atribuido, el cual está atribuido sobre un álgebra final de la signatura.

Definición 32. *(Grafo triple de tipos atribuido) Un* grafo triple de tipos atribuido *es un grafo triple atribuido* $TriATG = (TriTG, Z)$*, donde* Z *es el álgebra final de la signatura* $DSIG$ *con conjuntos portadores* $Z_s = \{s\}\ \forall s \in S_D$*.*

La figura C.10 muestra un grafo triple de tipos atribuido $TriATG = (TriTG, Z)$ para la definición de la sintaxis concreta y abstracta de los diagramas de secuencia UML. La signatura de datos $DSIG$ está definida como sigue:

```
DSIG = Char+String+
sorts : MessageType
opns :
synchronous: → MessageType
asynchronous: → MessageType
destroy: → MessageType
```

Esto es, el tipo $MessageType$ declara las constantes *synchronous*, *asynchronous* y *destroy*. Los tipos de datos usados para la atribución son $S'_D = \{String, MessageType\}$; $Char$ es un tipo auxiliar. El grafo destino en la parte superior del grafo triple corresponde a la sintaxis abstracta, que consiste en un subconjunto del meta-modelo UML. El grafo origen en la parte inferior corresponde a la sintaxis concreta. La sintaxis concreta declara objetos que pueden enlazarse con cajas de activación. Las cajas de activación pueden relacionarse con otras cajas por medio de líneas de vida y de mensajes, así como con los objetos a través de mensajes de creación. También existe un punto inicial con un mensaje de inicio. Por último, el grafo correspondencia en la parte intermedia relaciona los conceptos de los otros dos grafos.

La relación entre un grafo triple atribuido y un grafo triple de tipos atribuido se expresa mediante tuplas cuyo primer elemento es el grafo triple atribuido, y el segundo es un morfismo desde dicho grafo al grafo de tipos. Esta estructura se puede formalizar como una categoría slice.

Definición 33. *(Grafo triple tipado atribuido) Un* grafo triple tipado atribuido *(abreviado ATT-grafo) sobre* $TriATG$ *es un objeto* $TriTAG = (TriAG, t)$ *en la categoría slice* **TriAGraph/TriATG**, *donde* $TriAG = (TriG, D)$ *es un grafo triple atribuido y* $t\colon TriAG \to TriATG$ *es un morfismo de grafo triple atribuido denominado tipo de* $TriAG$*.*

Los morfismos entre grafos triples tipados atribuidos son como los morfismos entre grafos triples atribuidos donde, además, el tipo se preserva.

Definición 34. *(Morfismo de grafo triple tipado atribuido) Sean dos ATT-grafos* $TriTAG^i = (TriAG^i, t^i)$ *con* $i = 1, 2$ *sobre un grafo triple de tipos atribuido* $TriATG$*. Un* morfismo de grafo triple tipado atribuido *(abreviado ATT-morfismo)* $f\colon (TriAG^1, t^1) \to (TriAG^2, t^2)$

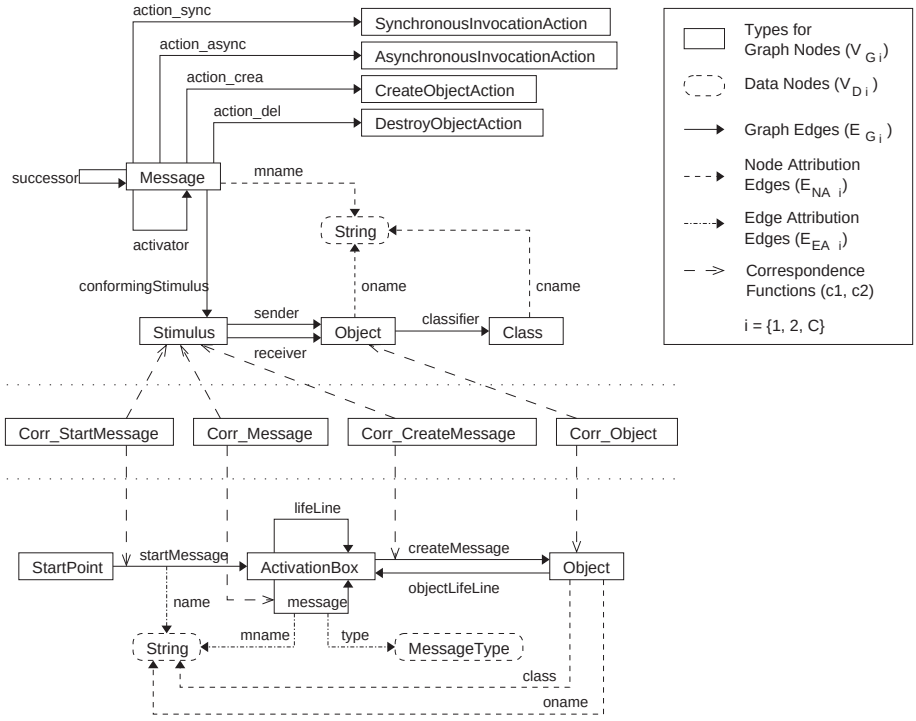


Figura C.10: *Ejemplo de grafo triple de tipos atribuido*

es un morfismo de grafo triple atribuido $f\colon TriAG^1 \to TriAG^2$ *tal que* $t^2 \circ f = t^1$*, como la figura C.11 muestra.*

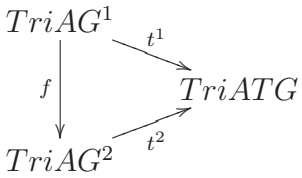


Figura C.11: *Condición para los morfismos de grafo triple tipado atribuido*

La figura C.12 muestra un ATT-grafo sobre el grafo de tipos de la figura C.10. Los nodos y relaciones están etiquetados con su tipo (siguiendo la notación de UML para instancias).

ATT-grafos y ATT-morfismos forman una categoría, donde los primeros son los objetos y los segundos los morfismos. Además, la composición de ATT-morfismos es asociativa y existe el morfismo identidad para cada objeto ATT-grafo de la categoría.

Definición 35. *(Categoría* $\mathbf{TriAGraphs_{TriATG}}$*) ATT-grafos sobre un grafo triple de tipos atribuido* $TriATG$ *y ATT-morfismos forman la categoría* $\mathbf{TriAGraphs_{TriATG}}$*.*

Demostración. Se deriva del hecho de que $\mathbf{TriAGraphs_{TriATG}}$ es una categoría slice. □

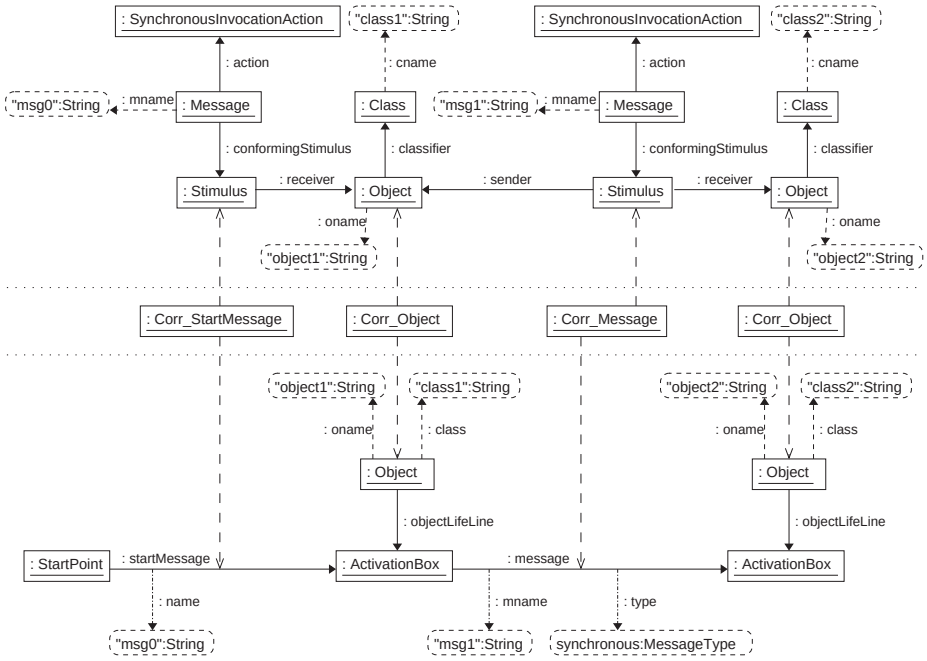


Figura C.12: *Grafo triple tipado atribuido, respecto al grafo triple de tipos atribuido de la figura C.10.*

C.2. Pushouts y pullbacks para grafos triples tipados atribuidos

En esta sección se muestra cómo construir pushouts y pullbacks en la nueva categoría $\mathbf{TriAGraphs_{TriATG}}$ (a partir de [61]). Los pushouts se necesitan para poder definir reglas de transformación de grafos (de ATT-grafos en este caso). Para ello basta con definirlos para una clase $\mathcal{M}$ de monomorfismos especiales, los cuales son inyectivos en la parte del grafo e isomorfismos en la parte de los datos. La razón es que las reglas de transformación en el enfoque *Double Pushout* (DPO) se modelan como $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ con l y r inyectivas (esto es, pertenecen a la clase $\mathcal{M}$ de monomorfismos). Por tanto es suficiente con definir los pushouts para los morfismos de $\mathcal{M}$. Por supuesto, $m\colon L \to G$ puede ser no inyectivo. Nótese que, como los TriE-grafos están formados por varios conjuntos, sus pushouts pueden construirse calculando el pushout de cada conjunto por separado.

En cuanto a los pullbacks, se utilizarán en la siguiente sección para demostrar que la categoría $\mathbf{TriAGraphs_{TriATG}}$ es HLR adhesiva (y que, por tanto, muchos de los resultados obtenidos para la transformación de grafos se pueden reutilizar para la transformación de ATT-grafos). Al igual que los pushouts, los pullbacks pueden construirse componente a componente.

A continuación se define la clase $\mathcal{M}$ de monomorfismos especiales en las categorías **TriAGraphs** y $\mathbf{TriAGraphs_{TriATG}}$.

Definición 36. *(Clase $\mathcal{M}$ de monomorfismos en* **TriAGraphs***) Un morfismo triple atribuido $f\colon TriAG^1 \to TriAG^2$ con $f = (f_{TriG} = (f^1, f^2, f^C), f_D)$ pertenece a la clase $\mathcal{M}$ si f_{TriG} es un morfismo de TriE-grafo inyectivo (esto es, si es inyectivo para cada componente de sus tres E-grafos), y además f_D es un isomorfismo de $DSIG-$algebra. Esto implica que $f^i_{V_D}$ es biyectivo.*

Definición 37. *(Clase $\mathcal{M}$ de monomorfismos en* $\mathbf{TriAGraphs_{TriATG}}$*) Un ATT-morfismo $f\colon (TriAG^1, t^1) \to (TriAG^2, t^2)$ pertenece a la clase $\mathcal{M}$ si $f\colon TriAG^1 \to TriAG^2$ pertenece a $\mathcal{M}$.*

Las clases $\mathcal{M}$ de monomorfismos en **TriAGraphs** y $\mathbf{TriAGraphs_{TriATG}}$ son cerradas bajo composición. A continuación se muestra cómo construir pushouts cuando uno de los morfismos pertenece a $\mathcal{M}$, lo que se hace básicamente calculando pushouts en **Sets**.

Construcción 38. *(Pushouts sobre $\mathcal{M}$-morfismos en* **TriAGraphs***) Sean dos morfismos triples atribuidos $f\colon TriAG^0 \to TriAG^1 \in \mathcal{M}$ y $g\colon TriAG^0 \to TriAG^2$. Un* pushout *$(TriAG^3, f', g')$ en* **TriAGraphs***, con $TriAG^k = (TriG^k, D^k)$, $TriG^k = (G^k_1, G^k_2, G^k_C, c^k_1, c^k_2)$, y $G^k_i = (V^k_{G_i}, V^k_{D_i}, E^k_{G_i}, E^k_{NA_i}, E^k_{EA_i}, (source^k_{j_i}, target^k_{j_i})_{j\in\{G,NA,EA\}})$ para $k \in \{0,1,2,3\}$, $i \in \{1,2,C\}$ se construye como sigue (véase figura C.13), donde $f' \in \mathcal{M}$ y cualquier otro pushout $TriAG'$ es isomorfo a $TriAG^3$:*

$$\begin{array}{ccc} TriAG^0 = (TriG^0, D^0) & \xrightarrow{f=((f^1,f^2,f^C),f_D)\in\mathcal{M}} & TriAG^1 = (TriG^1, D^1) \\ \Big\downarrow{\scriptstyle g=((g^1,g^2,g^C),g_D)} & = & \Big\downarrow{\scriptstyle g'=((g'^1,g'^2,g'^C),g'_D)} \\ TriAG^2 = (TriG^2, D^2) & \xrightarrow{f'=((f'^1,f'^2,f'^C),f'_D)\in\mathcal{M}} & TriAG^3 = (TriG^3, D^3) \end{array}$$

Figura C.13: *Construcción de pushouts/pullbacks en* **TriAGraphs**

1. $(V^3_{G_i}, f'^i_{V^3_{G_i}}, g'^i_{V^3_{G_i}})$ *es pushout de* $(V^0_{G_i}, f^i_{V^0_{G_i}}, g^i_{V^0_{G_i}})$ *en* **Sets** *para* $i \in \{1,2,C\}$.
2. $(E^3_{G_i}, f'^i_{E^3_{G_i}}, g'^i_{E^3_{G_i}})$ *es pushout de* $(E^0_{G_i}, f^i_{E^0_{G_i}}, g^i_{E^0_{G_i}})$ *en* **Sets** *para* $i \in \{1,2,C\}$.
3. $(E^3_{NA_i}, f'^i_{E^3_{NA_i}}, g'^i_{E^3_{NA_i}})$ *es pushout de* $(E^0_{NA_i}, f^i_{E^0_{NA_i}}, g^i_{E^0_{NA_i}})$ *en* **Sets** *para* $i \in \{1,2,C\}$.
4. $(E^3_{EA_i}, f'^i_{E^3_{EA_i}}, g'^i_{E^3_{EA_i}})$ *es pushout de* $(E^0_{EA_i}, f^i_{E^0_{EA_i}}, g^i_{E^0_{EA_i}})$ *en* **Sets** *para* $i \in \{1,2,C\}$.
5. $(V^3_{D_i}, f'^i_{V^3_{D_i}}, g'^i_{V^3_{D_i}}) = (V^2_{D_i}, id, g'^i_{V^2_{D_i}})$ *con* $g'^i_{V^2_{D_i}} = g^i_{V^2_{D_i}} \circ f^{-1,i}_{V^0_{D_i}}\colon V^1_{D_i} \to V^3_{D_i}$ *para* $i \in \{1,2,C\}$.
6. $(D^3, f'_D, g'_D) = (D^2, id, g'_D)$ *con* $g'_D = g_D \circ f_D^{-1}\colon D^1 \to D^3$.

7. *Las operaciones* $source^3_{j_i}$ *y* $target^3_{j_i}$ *(para* $j \in \{G, NA, EA\}$*,* $i \in \{1, 2, C\}$*) están unívocamente determinadas por los pushouts en (1)-(5).*

8. *Las funciones* c^3_1 *y* c^3_2 *están unívocamente determinadas por los pushouts en (1)-(4).*

Demostración. La construcción del pushout componente a componente lleva a la construcción de un grafo triple atribuido $TriAG^3$ bien formado, y a morfismos triples atribuidos f' y g' con $f' \in \mathcal{M}$. La propiedad universal de este pushout se deriva de la propiedad universal de los pushouts en **Sets**. Además, $(V^3_{D_i}, f'^i_{V^3_{D_i}}, g'^i_{V^3_{D_i}})$ y (D^3, f'_D, g'_D) son pushouts en **Sets** resp. $\mathbf{DSIG} - \mathbf{Alg}$. □

La figura C.14 muestra un ejemplo de pushout en la categoría **TriAGraphs**. Por simplicidad sólo se consideran nodos en V_G y E_G. En el ejemplo, $g \notin \mathcal{M}$ ya que los nodos 5 y 6 tienen como imagen el nodo 9, y por tanto los nodos 11 y 12 tienen como imagen el nodo 15. En consecuencia, la función g' no pertenece a $\mathcal{M}$.

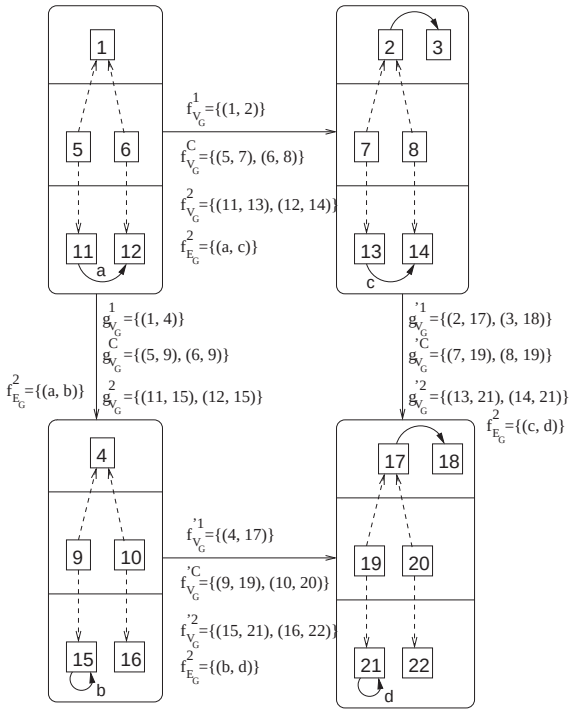


Figura C.14: *Ejemplo de pushout en la categoría* **TriAGraphs**

Construcción 39. *(Pushouts sobre* $\mathcal{M}$*-morfismos en* $\mathbf{TriAGraphs_{TriATG}}$*) Sean dos ATT-morfismos* $f\colon (TriAG^0, t^0) \to (TriAG^1, t^1) \in \mathcal{M}$ *y* $g\colon (TriAG^0, t^0) \to (TriAG^2, t^2)$*. El pushout* $((TriAG^3, t^3), f', g')$ *en* $\mathbf{TriAGraphs_{TriATG}}$ *se construye como el pushout* $(TriAG^3, f', g')$ *de* $f\colon TriAG^0 \to TriAG^1 \in \mathcal{M}$ *y* $g\colon TriAG^0 \to TriAG^2$ *en* **TriAGraphs**,

donde $t^3\colon TriAG^3 \to TriATG$ *está unívocamente determinado por las propiedades del pushout* $(TriAG^3, f', g')$ *en* **TriAGraphs**.

Demostración. Se deriva de la construcción de pushouts en las categorías slice. □

A continuación se muestra cómo construir pullbacks en los cuales uno de los morfismos pertenece a $\mathcal{M}$.

Construcción 40. *(Pullbacks sobre* $\mathcal{M}$*-morfismos en* **TriAGraphs***) Sean dos morfismos triples atribuidos* $f'\colon TriAG^2 \to TriAG^3 \in \mathcal{M}$ *y* $g'\colon TriAG^1 \to TriAG^3$*. Un pullback* $(TriAG^0, f, g)$ *en* **TriAGraphs**, *con* $TriAG^k = (TriG^k, D^k)$, $TriG^k = (G_1^k, G_2^k, G_C^k, c_1^k, c_2^k)$, *y* $G_i^k = (V_{G_i}^k, V_{D_i}^k, E_{G_i}^k, E_{NA_i}^k, E_{EA_i}^k, (source_{j_i}^k, target_{j_i}^k)_{j\in\{G,NA,EA\}})$ *para* $k \in \{0,1,2,3\}$, $i \in \{1,2,C\}$ *se construye como sigue (véase figura C.13), donde* $f' \in \mathcal{M}$ *y cualquier otro pullback* $TriAG'$ *es isomorfo a* $TriAG^0$*:*

1. $(V_{G_i}^0, f_{V_{G_i}^0}^i, g_{V_{G_i}^0}^i)$ *es pullback de* $(V_{G_i}^3, f_{V_{G_i}^3}'^i, g_{V_{G_i}^3}'^i)$ *en* **Sets** *para* $i \in \{1,2,C\}$.
2. $(E_{G_i}^0, f_{E_{G_i}^0}^i, g_{E_{G_i}^0}^i)$ *es pullback de* $(E_{G_i}^3, f_{E_{G_i}^3}'^i, g_{E_{G_i}^3}'^i)$ *en* **Sets** *para* $i \in \{1,2,C\}$.
3. $(E_{NA_i}^0, f_{E_{NA_i}^0}^i, g_{E_{NA_i}^0}^i)$ *es pullback de* $(E_{NA_i}^3, f_{E_{NA_i}^3}'^i, g_{E_{NA_i}^3}'^i)$ *en* **Sets** *para* $i \in \{1,2,C\}$.
4. $(E_{EA_i}^0, f_{E_{EA_i}^0}^i, g_{E_{EA_i}^0}^i)$ *es pullback de* $(E_{EA_i}^3, f_{E_{EA_i}^3}'^i, g_{E_{EA_i}^3}'^i)$ *en* **Sets** *para* $i \in \{1,2,C\}$.
5. $(V_{D_i}^0, f_{V_{D_i}^0}^i, g_{V_{D_i}^0}^i) = (V_{D_i}^1, id, g_{V_{D_i}^0}^i)$ *con* $g_{V_{D_i}^0}^i = f_{V_{D_i}^2}'^{-1,i} \circ g_{V_{D_i}^1}'^i : V_{D_i}^0 \to V_{D_i}^2$ *para* $i \in \{1,2,C\}$.
6. $(D^0, f_D, g_D) = (D^1, id, g_D)$ *con* $g_D = g'_D \circ f_D'^{-1}\colon D^0 \to D^2$.
7. *Las operaciones* $source_{j_i}^0$ *y* $target_{j_i}^0$ *(para* $j \in \{G, NA, EA\}$, $i \in \{1,2,C\}$*) están unívocamente determinadas por los pullbacks en (1)-(4).*
8. *Las funciones* c_1^0 *y* c_2^0 *están unívocamente determinadas por los pullbacks en (1)-(4).*

Demostración. La construcción del pullback componente a componente lleva a la construcción de un grafo triple atribuido $TriAG^0$ bien formado, y a morfismos triples atribuidos f y g con $f \in \mathcal{M}$. La propiedad universal de este pullback se deriva de la propiedad universal de los pullbacks en **Sets**. Además, $(V_{D_i}^0, f_{V_{D_i}^0}^i, g_{V_{D_i}^0}^i)$ y (D_i^0, f_D, g_D) son pullbacks en **Sets** resp. **DSIG** − **Alg**. □

Construcción 41. *(Pullbacks sobre* $\mathcal{M}$*-morfismos en* $\mathbf{TriAGraphs_{TriATG}}$*) Sean dos ATT-morfismos* $f'\colon (TriAG^2, t^2) \to (TriAG^3, t^3) \in \mathcal{M}$ *y* $g'\colon (TriAG^1, t^1) \to (TriAG^3, t^3)$*. El pullback* $((TriAG^0, t^0), f, g)$ *en* $\mathbf{TriAGraphs_{TriATG}}$ *se construye como el pullback* $(TriAG^0, f, g)$ *de* $f'\colon TriAG^2 \to TriAG^3 \in \mathcal{M}$ *y* $g'\colon TriAG^1 \to TriAG^3$ *en* **TriAGraphs**, *donde* $t^0\colon TriAG^0 \to TriATG$ *está unívocamente determinado por* $t^0 = t^1 \circ f = t^2 \circ g$.

Demostración. Se deriva de la construcción de pullbacks en las categorías slice. □

Construcción 42. *(Pushouts sobre $\mathcal{M}$-morfismos son pullbacks) Los pushouts sobre $\mathcal{M}$-morfismos en* **TriAGraphs** *y* $\mathbf{TriAGraphs}_{\mathbf{TriATG}}$ *son pullbacks.*

Demostración. Se debe a que los pushouts son inyectivos sobre funciones inyectivas en **Sets**. □

C.3. Grafos triples tipados atribuidos como categoría HLR adhesiva

En esta sección se da una formalización alternativa de grafos triples atribuidos en términos de una categoría coma. Con ello se pretende facilitar la demostración de que **TriAGraphs** y $\mathbf{TriAGraphs}_{\mathbf{TriATG}}$ son categorías HLR adhesivas. Esta demostración permitirá usar los principales resultados de la teoría de transformación de grafos sobre los objetos de ambas categorías, ya que dichos resultados se han definido para este tipo de categorías en [61].

La categoría **TriAGraphs** es isomorfa a una categoría coma $\mathbf{ComCat}(\mathbf{V_1}, \mathbf{V_2}; \mathbf{Id})$, donde V_1 y V_2 son funtores *forgetful*. El primero va de la categoría **TriEGraphs** a la categoría **Sets** y "olvida" la estructura de grafo triple, tomando el conjunto de valores de datos de uno de los grafos (ya que todos los conjuntos V_{D_i} son iguales). El segundo funtor V_2 va de la categoría **DSIG-Alg** a **Sets**, y coloca en un solo conjunto los elementos de los distintos conjuntos portadores para atribución (disyuntivamente). La categoría coma resultante tiene objetos $(TG, D, op\colon V_1(TG) \to V_2(D))$ que satisfacen $V_1(TG) = V_2(D)$ y $op = id$. Se puede demostrar que esta categoría, así como $\mathbf{TriAGraphs}_{\mathbf{TriATG}}$, es HLR adhesiva. A continuación se formaliza tal demostración, siguiendo el estilo de [61].

Construcción 43. *(Categoría coma para* **TriAGraphs***) La categoría* **TriAGraphs** *es isomorfa a una subcategoría* $\mathbf{ComCat}(\mathbf{V_1}, \mathbf{V_2}; \mathbf{Id})$ *de la categoría coma* $\mathbf{ComCat}(\mathbf{V_1}, \mathbf{V_2}; \mathbf{I})$ *con* $I = \{1\}$.

Sean V_1 y V_2 los funtores forgetful:

- $V_1\colon \mathbf{TriEGraphs} \to \mathbf{Set}$*, con* $V_1(TG) = V_{D_1}$ *y* $V_1(f = (f^1, f^2, f^C)) = f^1_{V_{D_1}}$*. Nótese que la elección de* V_{D_1} *en vez de* V_{D_2} *o* V_{D_C} *es irrelevante, ya que por construcción* $V_{D_1} = V_{D_2} = V_{D_C}$.
- $V_2\colon \mathbf{DSIG-Alg} \to \mathbf{Set}$*, con* $V_2(D) = \uplus_{s \in S'_D} D_s$ *y* $V_2(f_D) = \uplus_{s \in S'_D} f_{D_s}$.

$\mathbf{ComCat}(\mathbf{V_1}, \mathbf{V_2}; \mathbf{Id})$ *es la subcategoría de* $\mathbf{ComCat}(\mathbf{V_1}, \mathbf{V_2}; \mathbf{I})$ *con* $I = \{1\}$*, donde los objetos* $(TG, D, op\colon V_1(TG) \to V_2(D))$ *satisfacen que* $V_1(TG) = V_2(D)$ *y* $op = id$.

Demostración. Sea un grafo triple atribuido $TriAG = (TriG = (G_1, G_2, G_C, c_1, c_2), D)$ con $G_i = (V_{G_i}, V_{D_i}, E_{G_i}, E_{NA_i}, E_{EA_i}, (source_{j_i}, target_{j_i})_{j\in\{G,NA,EA\}}))$. Por la definición 29 tenemos que $\uplus_{s\in S'_D} D_s = V_{D_i}$ para $i \in \{1, 2, C\}$. Para los morfismos $f\colon TriAG^1 \to TriAG^2$ con $f = ((f^1, f^2, f^C), f_D)$ se tiene que (1) (2) y (3) en la figura C.15 conmutan. Si se toman los tipos de cada signatura y la parte de datos de G_1, se tiene que (4) en la figura C.16 conmuta.

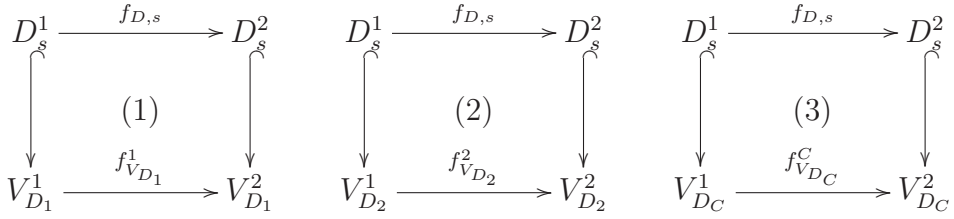


Figura C.15: *Condición para morfismos de grafo triple atribuido*

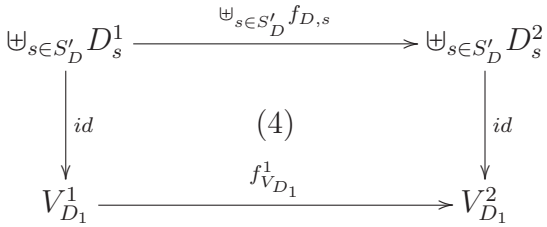


Figura C.16: *Condición para morfismos de grafo triple atribuido, grafo origen G_1. Ésta es la condición de compatibilidad de f y f_D en la construcción de la categoría coma*

Si se usa la condición $V_1(TG) = V_2(D)$ de la construcción de la categoría coma y la definición de los funtores V_1 y V_2, se tiene que $V^1_{D_1} = \uplus_{s\in S'_D} D_s$, que es exactamente la condición para grafos triples atribuidos (objetos de la categoría **TriAGraphs**, véase definición 29). Además, si se toma la definición de morfismo en la categoría coma se tiene que $f^1_{V_{D_1}} = \uplus_{s\in S'_D} f_{D_s}$. Ésta es la condición para morfismos de grafo triple atribuido que muestra la figura C.16, ya que $V_1(f = (f^1, f^2, f^C)) = f^1_{V_D}$ y $V_2(f_D) = \uplus_{s\in S'_D} f_{D_s}$. De ahí se deriva que $\mathbf{ComCat}(\mathbf{V_1}, \mathbf{V_2}; \mathbf{Id}) \cong \mathbf{TriAGraphs}$. □

La siguiente observación utiliza dos resultados de [61] (teoremas 4.13.3 y 4.13.4). El primero establece que si $(\mathbf{C}, \mathcal{M})$ es una categoría HLR adhesiva (débil), entonces para cada categoría $\mathbf{X}$ la categoría funtor ($[\mathbf{X}, \mathbf{C}]$, transformación $\mathcal{M}$-funtor) es una categoría HLR adhesiva (débil). Una transformación $\mathcal{M}$-funtor es una transformación natural $t\colon F \to G$ donde todos los morfismos $t_X\colon F(X) \to G(X)$ están en $\mathcal{M}$.

El segundo teorema establece que la categoría coma $(\mathbf{ComCat}(\mathbf{F}, \mathbf{G}; \mathcal{I}), \mathcal{M})$ con $\mathcal{M} = (\mathcal{M}_1 \times \mathcal{M}_2) \cap Mor_{ComCat(V_1,V_2;I)}$ es una categoría HLR adhesiva si F preserva los pushouts sobre $\mathcal{M}_1$-morfismos y G preserva los pullbacks generales. En el caso de categorías HLR

adhesivas débiles es suficiente con que F preserve los pushouts sobre $\mathcal{M}_1$-morfismos y G preserve los pullbacks sobre $\mathcal{M}_2$-morfismos.

Observación. (**TriAGraphs** es una categoría HLR adhesiva) Sea $\mathcal{M}_1$ la clase de morfismos de TriE-grafo inyectivos y $\mathcal{M}_2$ la clase de $DSIG-$isomorfismos. Entonces tenemos que $(\mathbf{TriEGraphs}, \mathcal{M}_1)$ es una categoría funtor sobre $(\mathbf{Sets}, \mathcal{M}_1)$[1] y por tanto una categoría HLR adhesiva. Por la misma razón $(\mathbf{DSIG-Alg}, \mathcal{M}_\mathbf{2})$ también es una categoría HLR adhesiva. Para que $\mathbf{ComCat}(\mathbf{V_1}, \mathbf{V_2}; \mathbf{I})$ con $I = 1$ y $\mathcal{M} = (\mathcal{M}_1 \times \mathcal{M}_2) \cap Mor_{ComCat(V_1,V_2;I)}$ sea una categoría adhesiva se tiene que demostrar que V_1 preserva los pushouts sobre los morfismos de $\mathcal{M}_1$ y V_2 preserva los pullbacks. En la categoría **TriEGraphs** los pushouts se construyen para cada componente en **Sets**; por tanto V_1 preserva los pushouts. Como se demuestra en [61], el funtor $F\colon \mathbf{DSIG-Alg} \to \mathbf{Set}$ preserva los pullbacks; por tanto V_2 preserva los pullbacks. Finalmente, también se puede demostrar que una elección especial de pushouts y pullbacks en **TriEGraphs** y **DSIG-Alg** llevan a pushouts y pullbacks en la subcategoría $\mathbf{ComCat}(\mathbf{V_1}, \mathbf{V_2}; \mathbf{Id})$, por lo que $(\mathbf{ComCat}(\mathbf{V_1}, \mathbf{V_2}; \mathbf{Id}), \mathcal{M})$ y $(\mathbf{TriAGraphs}, \mathcal{M})$ son categorías HLR adhesivas.

La siguiente observación usa el resultado del teorema 4.13.2 de [61], que establece que la categoría slice $(\mathbf{C/X}, \mathcal{M} \cap \mathbf{C/X})$ es una categoría HLR adhesiva donde $\mathcal{M} \cap C/X$ son monomorfismos en C/X (los monomorfismos en C también son monomorfismos en C/X, pero lo contrario no es necesariamente cierto).

Observación. ($\mathbf{TriAGraphs_{TriATG}}$ es una categoría HLR adhesiva) ($\mathbf{TriAGraphs_{TriATG}}$, $\mathcal{M}$) es una categoría slice de (**TriAGraphs**, $\mathcal{M}$) (donde $\mathcal{M}$ es la clase de los morfismos $f = ((f^1, f^2, f^C), f_D)$ con f^i inyectivos y siendo f_D un homomorfismo de $DSIG-$Algebra), y por tanto una categoría HLR adhesiva.

C.4. Transformación de grafos triples tipados atribuidos

La sección anterior demostró que $(\mathbf{TriAGraphs_{TriATG}}, \mathcal{M})$ es una categoría HLR adhesiva, por lo que podemos usar la teoría existente para transformación de grafos (generalizada en [61] para este tipo de categorías) sobre sus objetos. A modo ilustrativo, esta sección recoge los conceptos de transformación básicos (producción, derivación y gramática) para grafos triples tipados atribuidos. Además, las producciones se extienden con condiciones de aplicación. Para una relación exhaustiva de los resultados aplicables a sistemas HLR consúltese [61].

[1] Nótese que no se puede utilizar directamente la categoría esquema de la figura C.1, sino que han de añadirse elementos de correspondencia para relacionar nodos de un grafo con nodos del otro, nodos con enlaces, etc. Además, se han de incluir enlaces para cada combinación de estas correspondencias.

En el enfoque *Double Pushout* (DPO) las reglas se modelan mediante tres componentes: L, K y R. El componente L (o parte izquierda de la regla, LHS) representa los elementos que deben encontrarse en la estructura donde se aplica la regla (un grafo, un grafo triple, una red de Petri, etc.). El componente K contiene los elementos que se preservan al aplicar la regla. Por último, R (o parte derecha de la regla, RHS) contiene los elementos que deben reemplazar la parte identificada por L en la estructura a reescribir. Por tanto $L - K$ contiene los elementos que la regla borra, mientras que $R - K$ contiene los elementos que la regla crea. En este caso particular, L, K y R son ATT-grafos.

Definición 44. *(Regla triple) Sea un grafo triple de tipos atribuido $TriATG$ con signatura de datos $DSIG$. Una* regla de grafo triple tipado atribuido *(abreviado regla triple) $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ está formada por tres ATT-grafos L, K y R con una $DSIG$-álgebra común $T_{DSIG}(X)$ (la $DSIG$-álgebra de términos con variables X), y dos ATT-morfismos inyectivos $l\colon K \to L$ y $r\colon K \to R$.*

Observación. Puesto que l y r son ATT-morfismos inyectivos, la parte de la signatura $DSIG$ de l y r es la identidad en $T_{DSIG}(X)$.

Para aplicar una regla triple p a un ATT-grafo G (llamado ATT-grafo anfitrión) se debe encontrar una imagen de la LHS en el grafo. Esto es, debe existir un ATT-morfismo $m\colon L \to G$. Una vez encontrado tal morfismo la regla se aplica en dos pasos. En el primero se borran de G los elementos de $m(L - l(K))$, obteniendo el grafo D. En el segundo paso se añaden a D los elementos de $R - r(K)$, obteniendo de ese modo el grafo resultante H. Cada paso se modela mediante un pushout. Como se mostró en secciones anteriores, los pushouts en **TriAGraphs** y $\mathbf{TriAGraphs_{TriATG}}$ se calculan construyendo los pushouts de cada conjunto de los tres E-grafos del grafo triple.

Definición 45. *(Derivación directa de grafo triple) Sean una regla triple $p = (L \xleftarrow{l} K \xrightarrow{r} R)$, un ATT-grafo G, y un ATT-morfismo $m : L \to G$ (llamado* match *o imagen). Una* derivación directa de grafo triple *(abreviado derivación directa) $G \overset{p,m}{\Longrightarrow} H$ se calcula mediante dos pushouts en la categoría $\mathbf{TriAGraphs_{TriATG}}$, tal y como muestra la figura C.17, donde* (1) *y* (2) *son pushouts.*

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & (1) & {\scriptstyle d}\downarrow & (2) & \downarrow{\scriptstyle m^*} \\ G & \xleftarrow{l^*} & D & \xrightarrow{r^*} & H \end{array}$$

Figura C.17: *Derivación directa como construcción DPO*

La figura C.18 muestra un ejemplo de derivación directa. La regla conecta un objeto con su clase, para lo que crea una relación (etiquetada "7" en R y H). En la figura, los morfismos triples l, r, m, d, m^*, l^* y r^* están representados con números.

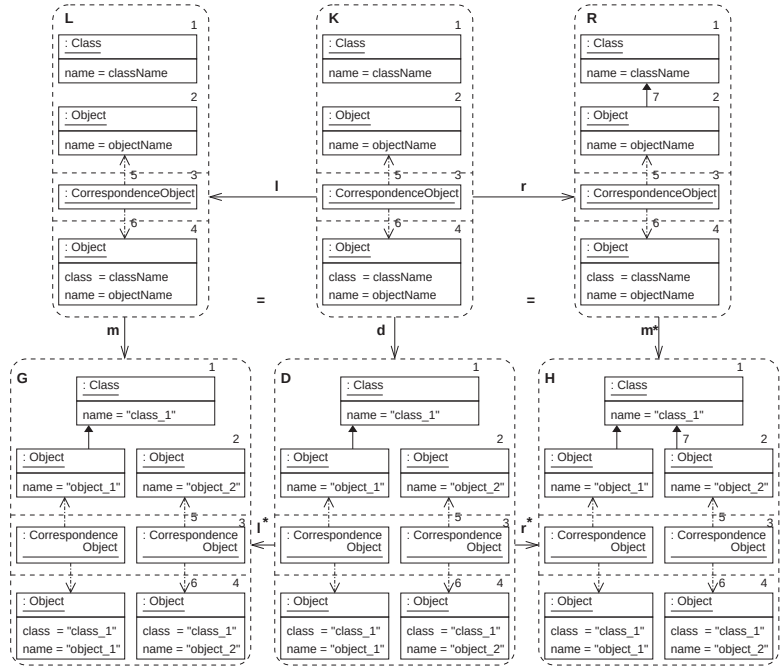


Figura C.18: *Ejemplo de derivación directa*

El primer pushout (1) calcula el grafo D, esto es, el complemento pushout. Para que el complemento pushout exista, además de cumplirse las condiciones de identificación y enlaces colgantes (también conocidas como condición de pegado) para cada grafo triple, se necesita una condición adicional para las funciones de correspondencia. A continuación se presenta la condición de pegado para las categorías **AGraphs** y $\mathbf{AGraphs_{ATG}}$, y posteriormente para las categorías **TriAGraphs** y $\mathbf{TriAGraphs_{TriATG}}$.

Definición 46. *(Condición de pegado en* **AGraphs** *y* $\mathbf{AGraphs_{ATG}}$*, tomadas de [61])*

1. *Sean una producción de grafo atribuido* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$*, un grafo atribuido* G*, y un morfismo* $m : L \to G$ *en la categoría* **AGraphs***, con* $X = (V_G^X, V_D^X, E_G^X, E_{NA}^X, E_{EA}^X, (source_j^X, target_j^X)_{j\in\{G,NA,EA\}}, D^X)$ *para todo* $X \in \{L, K, R, G\}$*.*

   - *Los puntos de pegado* GP *son aquellos elementos de grafo en* L *que* p *no borra, esto es,* $GP = l_{V_G}(V_G^K) \cup l_{E_G}(E_G^K) \cup l_{E_{NA}}(E_{NA}^K) \cup l_{E_{EA}}(E_{EA}^K)$*.*
   - *Los puntos de identificación* IP *son aquellos elementos de grafo en* L *identificados por* m*, esto es,* $IP = IP_{V_G} \cup IP_{E_G} \cup IP_{E_{NA}} \cup IP_{E_{EA}}$ *con:*
     $IP_{V_G} = \{a \in V_G^L | \exists a' \in V_G^L, a \neq a', m_{V_G}(a) = m_{V_G}(a')\}$
     $IP_{E_j} = \{a \in E_j^L | \exists a' \in E_j^L, a \neq a', m_{E_j}(a) = m_{E_j}(a')\}$*, para todo* $j \in \{G, NA, EA\}$*.*
   - *Los puntos colgantes* DP *son aquellos elementos de grafo en* L *cuyas imágenes son origen o destino de un elemento que no pertenece a* $m(L)$*, esto es,*

$DP = DP_{V_G} \cup DP_{E_G}$ *con:*
$DP_{V_G} = \{a \in V_G^L | (\exists a' \in E_{NA}^G - m_{E_{NA}}(E_{NA}^L),\ m_{V_G}(a) = source_{NA}^G(a')) \vee$
$(\exists a' \in E_G^G - m_{E_G}(E_G^L),\ m_{V_G}(a) = source_G^G(a') \vee m_{V_G}(a) = target_G^G(a'))\}$
$DP_{E_G} = \{a \in E_G^L | (\exists a' \in E_{EA}^G - m_{E_{EA}}(E_{EA}^L),\ m_{E_G}(a) = source_{EA}^G(a')\}$.
p y m satisfacen la condición de pegado en **AGraphs** *si todos los puntos colgantes y de identificación son puntos de pegado, esto es, si* $IP \cup DP \subseteq GP$.

2. *p y m en* $\mathbf{AGraphs_{ATG}}$ *satisfacen la condición de pegado en* $\mathbf{AGraphs_{ATG}}$ *si al considerarlos en* **AGraphs** *satisfacen la condición de pegado en* **AGraphs**.

Definición 47. *(Condición de pegado en* **TriAGraphs** *y* $\mathbf{TriAGraphs_{TriATG}}$*)*

1. *Sean un regla triple* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$*, un grafo triple atribuido* $TriAG = (TriG = (G_1, G_2, G_C, c_1^G, c_2^G), D)$*, y un morfismo* $m = (m_{TriG} = (m^1, m^2, m^C), m_D) : L \to G$ *en* **TriAGraphs**, *con* $X = (V_G^X, V_D^X, E_G^X, E_{NA}^X, E_{EA}^X, (source_j^X, target_j^X)_{j \in \{G,NA,EA\}})$ *para todo* $X \in \{L^i, K^i, R^i, G_i\}_{i \in \{1,2,C\}}$.

   - *Los puntos de pegado de correspondencia CGP son los elementos de grafo en* G_1 *y* G_2 *de L que p no borra, esto es,* $CGP = l_{V_G}^1(V_G^{K^1}) \cup l_{E_G}^1(E_G^{K^1}) \cup l_{V_G}^2(V_G^{K^2}) \cup l_{E_G}^2(E_G^{K^2})$.
   - *Los puntos colgantes de correspondencia CDP son los elementos de grafo en* G_1 *y* G_2 *de L, cuyas imágenes son destino de una función de correspondencia desde un elemento que no pertenece a* $m_{V_G}^C(L_{V_G}^C)$*, esto es,* $CDP = CDP_1 \bigcup CDP_2$ *para* $CDP_i = \{a \in L_{V_G}^i \cup L_{E_G}^i | \exists x \in G_{V_G}^C - m_{V_G}^C(L_{V_G}^C)$ *con* $c_i^G(x) = (m_{V_G}^i \uplus m_{E_G}^i)(a)\}$*, para* $i \in \{1, 2\}$

   p y m satisfacen la condición de pegado en **TriAGraphs** *si:*

   - *Los morfismos* $m_i' = (m^i, m_D) \colon (L^i, D) \to (G^i, D)$ *(para* $i \in \{1, 2, C\}$*) en* **AGraphs** *satisfacen la condición de pegado en* **AGraphs**.
   - *Todos los puntos colgantes de correspondencia son puntos de pegado de correspondencia, esto es,* $CDP \subseteq CGP$.

2. *p y m en* $\mathbf{TriAGraphs_{TriATG}}$ *satisfacen la condición de pegado en* $\mathbf{TriAGraphs_{TriATG}}$ *si al considerarlos en* **TriAGraphs** *satisfacen la condición de pegado en* **TriAGraphs**.

La condición de pegado de correspondencia establece que un elemento de los grafos anfitriones origen o destino no se puede borrar si es imagen de una función de correspondencia que sale de un elemento que no se borra.

Por otro lado, la segunda entrada de la definición 47.1 no necesita considerar los nodos del grafo correspondencia que son origen de una función de correspondencia. La razón es

que si uno de esos nodos se borra, la definición de la función de correspondencia para el nodo también se borra. La situación es similar a cuando se borra una relación en un grafo normal, donde la definición de las funciones *source* y *target* de la relación también se borran. Sin embargo, hay que tener cuidado al borrar el destino de una función de correspondencia ya que de otro modo las funciones podrían dejar de estar bien definidas.

Finalmente, la condición de identificación no necesita considerar la función de correspondencia. Si dos nodos del grafo correspondencia de L se identifican con un solo nodo de G, y la regla borra sólo uno de ellos, entonces la regla no se puede aplicar debido a la condición de identificación para el grafo correspondencia. La figura C.19 muestra un ejemplo donde los nodos B yC se identifican con el nodo BC del grafo correspondencia de G. La regla no puede aplicarse porque borra uno de los nodos y preserva el otro.

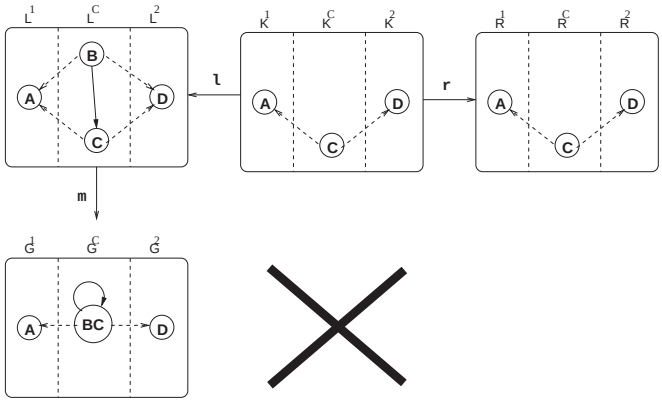


Figura C.19: *Aplicación de regla prohibida debido a la condición de identificación en el grafo correspondencia*

Observación. (Existencia y unicidad de grafos de contexto triples (tipados) atribuidos) Sean una regla de grafo triple (tipado) atribuido p, un grafo triple (tipado) atribuido G, y un morfismo $m : L \to G$. El grafo de contexto triple (tipado) atribuido D con $PO(1)$ existe en **TriAGraphs** (**TriAGraphs**$_{\mathbf{TriATG}}$), si y sólo si la condición de pegado se satisface en **TriAGraphs** (**TriAGraphs**$_{\mathbf{TriATG}}$). Si D existe, es único para isomorfismos.

$$\begin{array}{ccc} L & \xleftarrow{l \in \mathcal{M}} & K \\ {\scriptstyle m}\downarrow & (1) & \downarrow{\scriptstyle k} \\ G & \xleftarrow[f \in \mathcal{M}]{} & D \end{array}$$

Demostración. '⇒' Dado el $PO(1)$, las propiedades de la condición de pegado se derivan de las de los pushouts sobre $\mathcal{M}$-morfismos en **TriAGraphs** y **TriAGraphs**$_{\mathbf{TriATG}}$:

- Como los pushouts se construyen para cada uno de los componentes en **TriAGraphs** (**TriAGraphs**$_{\mathbf{TriATG}}$), la condición de pegado se cumple separadamente para cada grafo del grafo triple.

- Consideremos $v \in CDP$ y asumamos que $v \in L^i_{V_G}$ (esto es, v es un nodo) con $x \in G^C_{V_G} - m^C_{V_G}(L^C_{V_G})$ y $c^G_i(x) = m^i_{V_G}(v) = b$. En este caso, como G es un pushout, ha de existir un $x' \in V^D_{G_C}$ tal que $f^c_{V_G}(x') = x$. Más aún, ya que f es un morfismo válido ha de existir un $b' \in V^D_{G_i}$ con $c^D_i(x') = b'$ y $f^1_{V_G}(b') = b$. Esto implica que v no se borra. Para el caso en que v es una relación se sigue un razonamiento similar. En conclusión, $CDP \subseteq CGP$ y todos los puntos colgantes de correspondencia son también puntos de pegado de correspondencia.

'$\Leftarrow$' Si la condición de pegado se cumple podemos construir $D = (TriG^D = (G^D_1, G^D_2, G^D_C, c^D_1, c^D_2), D^D)$ con $G^D_i = (V^D_{G_i}, V^D_{D_i}, E^D_{G_i}, E^D_{NA_i}, E^D_{EA_i}, (source^D_{j,i}, target^D_{j,i})_{j\in\{G,NA,EA\}})$ $(i = \{1, 2, C\})$ con k, f y $type_D : D \to TriATG$ de la siguiente manera:

- $V^D_{G_i} = (V^D_{G_i} - m^i_{V_G}(L^i_{V_G})) \cup m^i_{V_G} \circ l^i_{V_G}(K^i_{V_G})$.
- $V^D_{D_i} = V^G_{D_i}$.
- $D^i_{E_j} = (G_{E_j} - m^i_{E_j}(L_{E_j})) \cup m \circ l(K^i_{E_j})$ para $j \in \{G, NA, EA\}$.
- $source^D_{G,i} = source^G_{G,i}|_{V^D_{G_i}}$, $target^D_{G,i} = target^G_{G,i}|_{V^D_{G_i}}$.
- $source^D_{j,i} = source^G_{j,i}|_{E^D_{j_i}}$, $target^D_{j,i} = target^G_{j,i}|_{E^D_{j_i}}$ para $j \in \{NA, EA\}$.
- $D^D = D^G$.
- $c^D_i(x) = c^G_i(x)$, $\forall x \in D^C_{V_G}$, para $i = \{1, 2\}$.
- $k(x) = m(l(x))$ para todo x en K.
- f inclusión.
- $type_D = type_G|_D$.

□

Una derivación triple es una secuencia de cero o más derivaciones triples directas, lo que se denota como $G_0 \Rightarrow^* G_n$. Una gramática de grafos triple está formada por un conjunto de reglas triples y un ATT-grafo inicial (junto con una signatura de datos y un grafo de tipos). El lenguaje de la gramática triple consiste en todos los ATT-grafos que se pueden obtener mediante derivaciones, empezando desde el ATT-grafo inicial.

Definición 48. *(Gramática y lenguaje de grafos triples) Una* gramática de grafos triples *$TGG = (DSIG, TriATG, P, TriAS)$ está compuesta por una signatura de datos $DSIG$, un grafo triple de tipos atribuido $TriATG$, un conjunto P de reglas triples, y un ATT-grafo inicial $TriAS$ tipado sobre $TriATG$. El* lenguaje *que genera TGG se define como $L(TGG) = \{TriAG | TriAS \Rightarrow^* TriAG\}$.*

A continuación se provee a las reglas triples con condiciones de aplicación, siguiendo el mismo enfoque que en [93]. Para ello se define el concepto de restricción condicional sobre ATT-grafos, de tal modo que una condición de aplicación se exprese como una restricción condicional sobre el componente L de la regla triple.

Definición 49. *(Restricción condicional triple) Una* restricción condicional triple $cc = (x\colon L \to X, A)$ *sobre un ATT-grafo L consiste en un ATT-morfismo x y un conjunto $A = \{y_j\colon X \to Y_j\}$ de ATT-morfismos. Un ATT-morfismo $m\colon L \to G$ satisface una restricción cc sobre L, escrito $m \models_L cc$, si y sólo si $\forall n\colon X \to G$ con $n \circ x = m$ $\exists o\colon Y_j \to G$ (donde $y_j\colon X \to Y_j \in A$) tal que $o \circ y_j = n$ (véase figura C.20).*

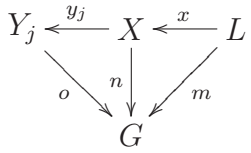


Figura C.20: *Restricción condicional triple satisfecha por m*

El morfismo m satisface la restricción si no existe ninguna imagen de X en G o, en caso de que exista, si también se encuentra la imagen de algún Y_j. Si el conjunto A es el conjunto vacío la restricción se denomina condición de aplicación negativa (NAC), y en ese caso la existencia de un ATT-morfismo n implica que $m \not\models_L cc$. Los morfismos x e y_j son totales, pero gráficamente se usa una notación abreviada. En concreto, el subgrafo de L (resp. X) que no tiene imagen en X (resp. Y_j) se copia isomórficamente en X (resp. Y_j) y se enlaza con los elementos que corresponda.

A cada regla triple se le asigna un conjunto AC de restricciones condicionales triples (llamadas *condición de aplicación*). Para poder aplicar una regla en cierto morfismo m, éste tiene que satisfacer todas las condiciones de aplicación del conjunto. La figura C.21 muestra como ejemplo una regla triple con NAC (el conjunto A en la condición de aplicación está vacío). Siguiendo la notación abreviada, la NAC sólo contiene los elementos adicionales que no aparecen en la LHS y su contexto. El componente K de la regla no aparece explícitamente, pero contendría aquellos elementos etiquetados con los mismos números en la LHS y la RHS. La regla crea un objeto en el grafo destino (parte superior de la regla) con el mismo nombre que uno existente en el grafo origen. La NAC hace que la regla no se aplique si ya existe tal objeto en el grafo destino.

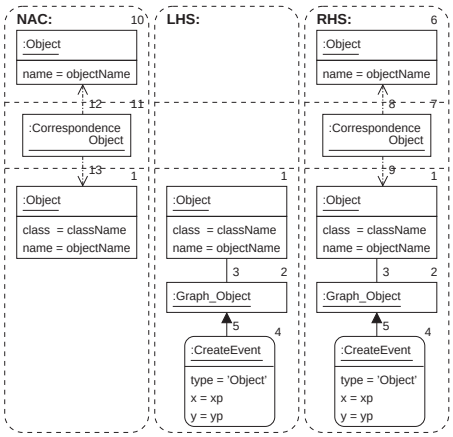


Figura C.21: *Ejemplo de regla triple con condición de aplicación negativa*

Apéndice D

Herencia en transformación de grafos triples tipados atribuidos

Este apéndice extiende los grafos triples y sistemas de transformación de grafos triples presentados en el apéndice C con herencia. La extensión se basa en la que [60, 45] realizan para grafos, pero adaptada a grafos triples y permitiendo herencia no sólo de nodos, sino también de relaciones.

Con este propósito, la definición de grafo triple de tipos se extiende para incluir relaciones de herencia entre dos nodos o relaciones, de modo similar a como se realiza en los diagramas de clases UML. La herencia de nodos permite a un nodo heredar las relaciones y atributos de sus nodos padre (y demás ancestros). La herencia de relaciones permite a una relación heredar los atributos de sus relaciones padre (y demás ancestros).

Una vez así definidos los grafos triples con herencia, se permite a las reglas triples contener elementos (nodos y relaciones) cuyo tipo tiene un conjunto de subtipos. Si estos elementos aparecen en la parte izquierda de la regla, cualquier nodo del grafo anfitrión que tenga su tipo o uno de sus subtipos es una imagen válida del elemento. Es decir, una regla extendida con herencia es equivalente a un conjunto de reglas (llamadas reglas concretas) resultantes de sustituir cada nodo y relación de grafo por sí mismo y cada uno de sus subtipos. De este modo se consigue tener una notación compacta de un conjunto de reglas.

La inclusión de herencia en las reglas de transformación de grafos triples es otra aportación adicional que no presentan las gramáticas de grafos triples [162]. Su formalización se presenta como un apéndice separado ya que se ha querido mantener diferenciado el núcleo con la teoría básica y necesaria para la transformación de grafos triples (apéndice C), de sus extensiones (presente apéndice). Por la misma razón, el apéndice da una idea intuitiva de la formalización y presenta sólo las definiciones de los conceptos más relevantes. El lector interesado puede encontrar el resto de detalles de la formalización en [84].

D.1. Grafos triples tipados atribuidos con herencia

Los grafos triples de tipos con herencia se definen como los grafos triples de tipos usuales, con dos grafos adicionales para las jerarquías de nodos y relaciones, y dos conjuntos de nodos y relaciones abstractos. Por razones técnicas relacionadas con la herencia de

la función de correspondencia, los nodos del grafo correspondencia no admiten herencia múltiple. Siguiendo el mismo enfoque que en [178], una relación puede heredar de otra únicamente si sus nodos fuente y destino son hijos de los nodos fuente y destino de la relación de la que hereda. Para asegurar esto se utiliza la noción de *clan*, que consiste en una función que devuelve el conjunto de nodos o relaciones hijo de un elemento dado, incluido él mismo (véase definición 51).

Definición 50. *(Grafo triple de tipos atribuido con herencia) Un* grafo triple de tipos atribuido con herencia *(abreviado meta-modelo triple)* $TriATGI = (TriATG, (VI_i, EI_i, AV_i, AE_i)_{i\in\{1,2,C\}})$*, consiste en:*

- *Un grafo triple de tipos atribuido* $TriATG = (TriTG, Z)$.
- *Tres grafos de herencia de nodos* $VI_i = (VI_V^i, VI_E^i, vs^i\colon VI_E^i \to VI_V^i, vt^i\colon VI_E^i \to VI_V^i)$*, con* $VI_V^i = V_G^i$ *para* $i \in \{1, 2, C\}$*. No se permite herencia múltiple en el grafo correspondencia, por lo que* $\forall n \in VI_V^C$, $|\{e \in VI_E^C | vs^C(e) = n\}| \leq 1$.
- *Tres grafos de herencia de relaciones* $EI_i = (EI_V^i, EI_E^i, es^i\colon EI_E^i \to EI_V^i, et^i\colon EI_E^i \to EI_V^i)$*, con* $EI_V^i = E_G^i$ *para* $i \in \{1, 2, C\}$*. Además* $\forall e, e' \in EI_V^i, x \in EI_E^i$ *tal que* $es^i(x) = e'$ *y* $et^i(x) = e$ *(esto es, si* e' *hereda de* e*), se tiene que* $source_{G_i}(e') \in clan_{VI^i}(source_{G_i}(e))$ *y* $target_{G_i}(e') \in clan_{VI^i}(target_{G_i}(e))$.
- *Tres conjuntos* $AV_i \subseteq VI_V^i$ *para* $i = \{1, 2, C\}$*, llamados nodos abstractos.*
- *Tres conjuntos* $AE_i \subseteq EI_V^i$ *para* $i = \{1, 2, C\}$*, llamados relaciones abstractas.*

La figura D.1 muestra un ejemplo de meta-modelo triple, el cual extiende el grafo triple de tipos atribuido de la figura C.10 con herencia. El meta-modelo triple relaciona dos meta-modelos con la sintaxis abstracta (parte superior) y concreta (parte inferior) de los diagramas de secuencia UML. Para facilitar la lectura, los grafos TG_i, el grafo de herencia de nodos VI_i y el grafo de herencia de relaciones EI_i de la figura se muestran colapsados en un único grafo. Las relaciones de los grafos de herencia se representan por una línea continua con una flecha hueca que apunta al tipo padre, al estilo usual en la notación UML, y los elementos en AV_i y AE_i se muestran en itálica. Las relaciones de composición que aparecen en el meta-modelo (con un diamante negro en el extremo) se tratan como cualquier otra relación de E_G^i.

Para poder usar la teoría desarrollada en el apéndice C sobre meta-modelos triples, se "aplana" su jerarquía haciendo explícito el significado semántico de la herencia, y obteniendo de ese modo un grafo triple de tipos atribuido usual que es equivalente. La operación de aplanado copia explícitamente los elementos a heredar a todos los descendientes (relaciones y atributos en el caso de nodos, y atributos en el caso de relaciones). En la teoría desarrollada hasta la fecha no se contempla la sobreescritura de atributos, pero sí la sobreescritura

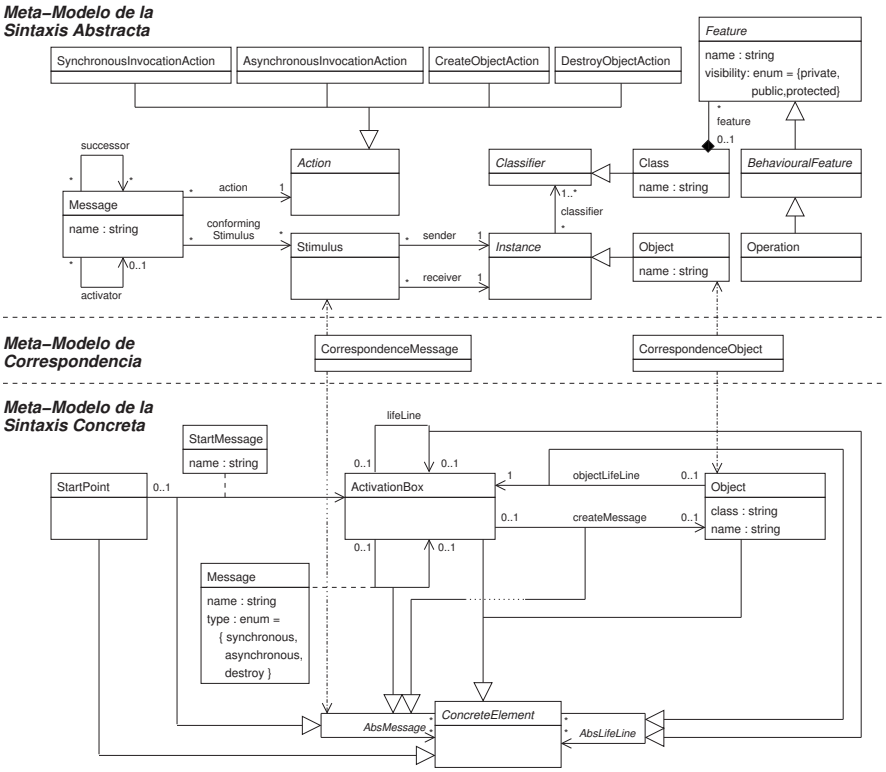


Figura D.1: *Ejemplo de meta-modelo triple*

de funciones de correspondencia por los nodos del grafo correspondencia. De este modo, si una función de correspondencia está indefinida para un nodo del grafo correspondencia, entonces su valor se obtiene del nodo más cercano en la jerarquía de herencia (de nodos) para el que la función está definida.

En realidad, esta versión aplanada del grafo de tipos no se utiliza para formalizar el tipo de un grafo triple, sino que su tipo se especifica directamente sobre el meta-modelo triple usando una clase de morfismo especial denominado *morfismo triple de clan*. De hecho, este es enfoque seguido en [45]. Estos morfismos de clan tienen en cuenta las relaciones de herencia de nodos y relaciones, y se corresponden unívocamente con el tipo proporcionado por el grafo de tipos aplanado. A continuación se formaliza el concepto de clan de herencia; el lector interesado puede consultar la definición del resto de conceptos en [84].

Definición 51. *(Clan de herencia de nodos y de relaciones) Sea un meta-modelo triple* $TriATGI = (TriATG, (VI_i, EI_i, AV_i, AE_i)_{i\in\{1,2,C\}})$. *El* clan de herencia de nodos *para cada nodo* $n \in VI_V^i$ *se define como* $clan_{VI^i}(n) = \{n' \in VI_V^i \mid \exists\ path\ n' \xrightarrow{*} n \in VI_i\} \subseteq VI_V^i$ *con* $n \in clan_{VI^i}(n)$. *Igualmente, para cada relación* $e \in EI_V^i$, *su* clan de herencia de relaciones *se define como* $clan_{EI^i}(e) = \{e' \in EI_V^i \mid \exists\ path\ e' \xrightarrow{*} e \in EI_i\} \subseteq EI_V^i$ *con* $e \in clan_{EI^i}(e)$.

Por ejemplo, el clan de herencia del nodo $ConcreteElement$ en el meta-modelo de la figura D.1 es $clan_{VI^1}(ConcreteElement) = \{ConcreteElement, ActivationBox, StartPoint, Object\}$. En el mismo meta-modelo, el clan de herencia de relaciones de $AbsMessage$ es $clan_{EI^1}(AbsMessage) = \{AbsMessage, createMessage, Message, StartMessage\}$.

D.2. Transformación de grafos triples tipados atribuidos con herencia

En este apartado las reglas triples se extienden para incluir el concepto de herencia, dando como resultado la definición de un nuevo tipo de regla denominado *regla triple extendida con herencia*, o regla IE-triple. Los nodos y relaciones en las reglas IE-triples pueden tener como tipo cualquiera de los definidos en el meta-modelo triple, el cual se refina posteriormente al conjunto de sus subtipos. Una regla IE-triple es equivalente a todas las sustituciones válidas de cada uno de sus nodos y relaciones por cualquiera de los tipos concretos de los nodos y relaciones en su clan de herencia. Si el conjunto de reglas equivalente tiene una cardinalidad mayor que 1, entonces la regla IE-triple se denomina *meta-regla IE-triple*. A continuación se especifica qué es un refinamiento de tipos, y se da la definición de regla IE-triple.

Definición 52. *(Refinamiento de tipo) Sea el grafo triple atribuido $TriAG = (TriG, D)$, donde $TriG = (G_1, G_2, G_C, c_1, c_2)$ es un grafo triple con $G_i = (V^G_{G_i}, V^G_{D_i}, E^G_{G_i}, E^G_{NA_i}, E^G_{EA_i}, (source^G_{j_i}, target^G_{j_i})_{j\in\{G,NA,EA\}})_{i\in\{1,2,C\}}$, y dos morfismos de clan $type\colon TriG \to TriATGI$ y $type'\colon TriG \to TriATGI$; $type'$ se llama* refinamiento de tipo *de $type$, escrito $type' \leq type$*[1] *si:*

- $type'^i_{V_G}(n) \in clan_{VI^i}(type^i_{V_G}(n))$, $\forall n \in V^G_{G_i}$, *para* $i \in \{1, 2, C\}$.
- $type'^i_{E_G}(n) \in clan_{EI^i}(type^i_{E_G}(n))$, $\forall n \in E^G_{G_i}$, *para* $i \in \{1, 2, C\}$.
- $type'^i_X = type^i_X$, *para* $X \in \{V_D, E_{NA}, E_{EA}\}, i \in \{1, 2, C\}$.
- $type'_D = type_D$.

Definición 53. *(Regla triple extendida con herencia) Una* regla triple extendida con herencia *(abreviado regla IE-triple) es una regla triple cuyo tipo está definido por un meta-modelo triple $TriATGI = (TriATG, (VI_i, EI_i, AV_i, AE_i)_{i\in\{1,2,C\}})$. Se define como una tripleta $p = (L \xleftarrow{l} K \xrightarrow{r} R, type, AC)$ donde el primer elemento es una regla de grafo triple atribuido (l y r son morfismos de grafo triple atribuido); $type = (type_i\colon i \to TriATGI)_{i\in\{L,K,R\}}$ es una tripleta de morfismos de clan, uno por cada parte de la regla triple; y $AC = \{cc_i =$*

[1] Decimos que $type'$ es más concreto que $type$.

$(x_i\colon L \to X_i, type_{X_i}, A_i = \{(y_{ij}\colon X_i \to Y_{ij}, type_{Y_{ij}})\})\}$ *es un conjunto de condiciones de aplicación donde* $type_{X_i}\colon X_i \to TriATGI$ *y* $type_{Y_{ij}}\colon Y_{ij} \to TriATGI$ *son morfismos de clan. Además se deben cumplir las siguientes condiciones:*

- $type_L \circ l = type_K = type_R \circ r$ *(el tipo de los elementos que la regla preserva es el mismo en L, K y R).*
- $type^i_{R,V_G}(V'^R_{G_i}) \cap AV_i = \emptyset$, *donde* $V'^R_{G_i} := V^R_{G_i} - r^i_{V_G}(V^K_{G_i})$ *para* $i \in \{1, 2, C\}$ *(ningún nodo nuevo en R tiene tipo abstracto).*
- $type^i_{R,E_G}(E'^R_{G_i}) \cap AE_i = \emptyset$, *donde* $E'^R_{G_i} := E^R_{G_i} - r^i_{E_G}(E^K_{G_i})$ *para* $i \in \{1, 2, C\}$ *(ninguna relación en R tiene tipo abstracto).*
- $type_{Y_{ij}} \circ y_{ij} \leq type_{X_i}$ *y* $type_{X_i} \circ x_i \leq type_L$ *para todo* $cc_i \in AC$ *(el tipo de* Y_{ij} *es más concreto que el tipo de* X_i*, y éste a su vez es más concreto que el tipo de L).*
- $type^i_{L,V_G} \circ c^L_i \circ l^C_{V^C_G}|_{nodes^K_i} = type^i_{K,V_G} \circ c^K_i|_{nodes^K_i} = type^i_{R,V_G} \circ c^R_i \circ r^C_{V^C_G}|_{nodes^K_i}$ *para* $i = 1, 2$ *donde* c^K_i, c^L_i *y* c^R_i *son funciones de correspondencia de K, L y R (el tipo de los nodos destino de las funciones de correspondencia en K es el mismo en L y R).*
- $type^i_{L,E_G} \circ c^L_i \circ l^C_{V^C_G}|_{edges^K_i} = type^i_{K,E_G} \circ c^K_i|_{edges^K_i} = type^i_{R,E_G} \circ c^R_i \circ r^C_{V^C_G}|_{edges^K_i}$ *para* $i = 1, 2$ *(el tipo de las relaciones destino de las funciones de correspondencia en K es el mismo en L y R).*
- $c^L_i \circ l^C_{V^C_G}|_{undef^K_i} = c^K_i|_{undef^K_i} = c^R_i \circ r^C_{V^C_G}|_{undef^K_i}$ *para* $i = 1, 2$ *(las funciones de correspondencia indefinidas en K también están indefinidas en L y R).*
- *La parte de tipos de datos en L, K, R,* X_i *y* Y_{ij} *es* $T_{DSIG}(X)$*, el álgebra de términos DSIG con variables X, y* $l_D, r_D, x_{i_D}, y_{ij_D}$ *son identidades (se preservan los datos).*

La fila superior de la figura D.2 muestra un ejemplo de meta-regla IE-triple. La regla identifica en la sintaxis concreta (parte inferior de la regla) un mensaje que activa otro mensaje, y crea en la sintaxis abstracta (parte superior) la relación *activator* apropiada. Esta meta-regla es equivalente a cuatro reglas concretas, ya que el nodo 7 puede tener los tipos concretos *StartPoint* o *ActivationBox*, el nodo 8 tiene que ser un *ActivationBox*, y el nodo 9 puede ser un *Object* o un *ActivationBox*. Por tanto, hay cuatro combinaciones posibles, donde el tipo de las relaciones lo determina la elección de los tipos de nodo.

Para aplicar una meta-regla IE-triple a un grafo triple primero se debe encontrar una imagen de la estructura de su parte izquierda (sin considerar los tipos) en el grafo. El tipo de la imagen debe ser más concreto que el de la parte izquierda de la regla. Además, el tipo de los nodos y relaciones que son destino de funciones de correspondencia en el grafo

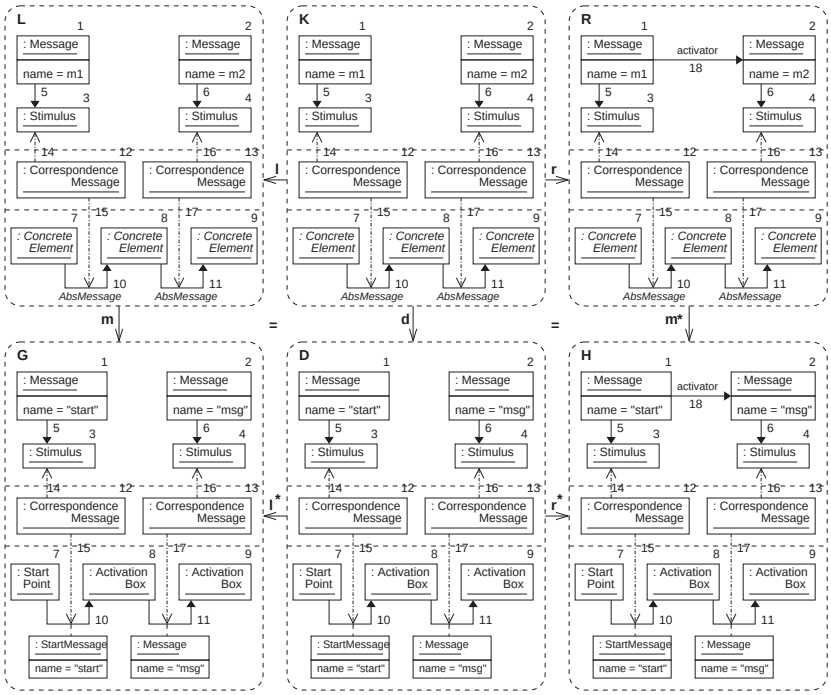


Figura D.2: *Ejemplo de meta-regla triple y derivación*

también debe ser más concreto que el especificado en la regla. Por último, se tienen que satisfacer las condiciones de aplicación.

Una derivación directa se realiza calculando primero el doble pushout en la categoría **TriAGraphs**. A continuación, al grafo resultante se le añade el tipo del siguiente modo: los elementos que la regla preserva no cambian su tipo, y los elementos que crea toman el tipo especificado en R (recuérdese que los elementos que una regla añade tienen tipo concreto). La figura D.2 muestra un ejemplo de derivación directa, donde los elementos abstractos 7, 8, 9, 10 y 11 de la regla toman los tipos concretos *StartPoint*, *ActivationBox*, *ActivationBox*, *StartMessage* y *Message* en el grafo G.

La aplicación de una meta-regla equivale a la aplicación de una de sus reglas concretas. Se puede demostrar que el lenguaje generado por una gramática de grafos triples extendida con herencia es el mismo que el generado por una gramática de grafos triples sin herencia. Esta segunda gramática usaría como grafo de tipos el resultante de aplanar el meta-modelo triple, y como reglas las reglas concretas de cada meta-regla de la primera gramática. Los detalles de esta demostración pueden consultarse en [84].

Bibliografía

[1] S. Abrahao, N. Condori-Fernández, L. Olsina, and O. Pastor, *Defining and validating metrics for navigational models*, in METRICS '03: Proceedings of the 9^{th} IEEE International Software Metrics Symposium, 2003, pp. 200–210.

[2] ACL2. `http://www.cs.utexas.edu/users/moore/acl2/`.

[3] ADM, *Architecture-Driven Modernization*. `http://adm.omg.org`.

[4] ——, *Architecture-Driven Modernization, request for proposal of metrics software metamodel*. `http://www.omg.org/docs/admtf/06-09-01.pdf`.

[5] AGG, *The Attributed Graph Grammar system*. `http://tfs.cs.tu-berlin.de/agg/`.

[6] A. Agrawal, G. Karsai, S. Neema, F. Shi, and A. Vizhanyo, *The design of a language for model transformations*, Journal on Software and Systems Modeling, 5 (2006), pp. 261–288.

[7] R. Alur and R. P. Kurshan, *Timing analysis in COSPAN*, in Hybrid systems III: Verification and control, vol. 1066, Springer, 1995, pp. 220–231.

[8] S. W. Ambler, *Process patterns: Building large-scale systems using object technology*, Cambridge University Press, 1998.

[9] AndroMDA. `http://galaxy.andromda.org/`.

[10] ARCE, *Aplicación en Red para Casos de Emergencia*. `http://arce.dei.inf.uc3m.es`.

[11] ArgoUML. `http://argouml.tigris.org/`.

[12] C. Atkinson and T. Kühne, *Rearchitecting the UML infrastructure*, ACM Transactions on Modeling and Computer Simulation, 12 (2002), pp. 290–321.

[13] ——, *Model-driven development: A metamodeling foundation*, IEEE Software, 20 (2003), pp. 36–41.

[14] ATL, *Atlas Transformation Language*. `http://www.sciences.univ-nantes.fr/lina/atl/`.

[15] J. C. M. Baeten and W. P. Weijland, *Process algebra*, Cambridge University Press, 1990.

[16] P. Baldan, A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi, *Handbook of graph grammars and computing by graph transformation: Volume III. Concurrency, parallelism, and distribution*, World Scientific Publishing Co. Inc., 1999, ch. Concurrent semantics of algebraic graph transformations, pp. 107–187.

[17] A. Balogh and D. Varró, *Advanced model transformation language constructs in the VIATRA2 framework*, in SAC '06: Proceedings of the 21^{st} ACM Symposium on Applied Computing, ACM Press, 2006, pp. 1280–1287.

[18] R. Bardohl, M. Niemann, and M. Schwarze, *GenGEd: A development environment for visual languages*, in AGTIVE '99: Proceedings of the 1^{st} International Workshop on Applications of Graph Transformations with Industrial Relevance, vol. 1779 of Lecture Notes in Computer Science, Springer-Verlag, 2000, pp. 233–240.

[19] L. Baresi and M. Pezzé, *Formal interpreters for diagram notations*, ACM Transactions on Software Engineering and Methodology, 14 (2005), pp. 42–84.

[20] V. Basili, G. Caldiera, and H. D. Rombach, *Encyclopaedia of software engineering, vol.1*, John Wiley & Sons, 1994, ch. Goal/Question/Metric paradigm, pp. 528–532.

[21] I. Beer, S. Ben-David, C. Eisner, and A. Landver, *RuleBase: An industry-oriented formal verification tool*, in DAC '96: Proceedings of the 33^{rd} Annual Conference on Design Automation, ACM Press, 1996, pp. 655–660.

[22] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore, R. Wade, and V. Sunderam, *HeNCE: Graphical development tools for network-based concurrent computing*, in SHPCC '92: Proceedings of the 1992 Scalable High Performance Computing Conference, 1992, pp. 129–136.

[23] D. M. Berry, *Formal methods: The very idea, some thoughts about why they work when they work*, Science of Computer Programming, 42 (2002), pp. 11–27.

[24] A. Bondavalli, D. Latella, M. D. Cin, and A. Pataricza, *High-level integrated design environment for dependability (HIDE)*, in WORDS '99: Proceedings of the 5^{th} International Workshop on Object-Oriented Real-Time Dependable Systems, IEEE Computer Society, 1999, pp. 87–92.

[25] A. Boronat, J. A. Carsí, and I. Ramos, *Algebraic specification of a model transformation engine*, in FASE '06: Proceedings of the 9^{th} International Conference on Fundamental Approaches to Software Engineering, L. Baresi and R. Heckel, eds., vol. 3922 of Lecture Notes in Computer Science, Springer, 2006, pp. 262–277.

[26] R. A. Botafogo, E. Rivlin, and B. Shneiderman, *Structural analysis of hypertexts: Identifying hierarchies and useful metrics*, ACM Transactions on Information Systems, 10 (1992), pp. 142–180.

[27] P. Bottoni, J. de Lara, and E. Guerra, *Action patterns for the specification of the execution semantics of visual languages*, in VLHCC '07: Proceedings of the 2007 IEEE Symposium on Visual Languages - Human Centric Computing, IEEE Computer Society, 2007, pp. 163–170.

[28] P. Bottoni and A. Grau, *A suite of metamodels as a basis for a classification of visual languages*, in VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing, IEEE Computer Society, 2004, pp. 83–90.

[29] P. Bottoni, E. Guerra, and J. de Lara, *Metamodel-based definition of interaction with visual environments*, in MDDAUI '06: Proceedings of the 2^{nd} International Workshop on Model Driven Development of Advanced User Interfaces (Satellite Event of MoDELS 2006), vol. 214 of CEUR Workshop Proceedings, CEUR-WS.org, 2006.

[30] H. Bowman, M. W. A. Steen, E. A. Boiten, and J. Derrick, *A formal framework for viewpoint consistency*, Formal Methods in System Design, 21 (2002), pp. 111–166.

[31] R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa, *VIS*, in FMCAD '96: Proceedings of the 1^{st} International Conference on Formal Methods in Computer-Aided Design, Springer-Verlag, 1996, pp. 248–256.

[32] BridgePoint. `http://www.mentor.com/products/sm/uml_suite/index.cfm`.

[33] R. E. Bryant, *Symbolic boolean manipulation with ordered binary decision diagrams*, tech. report, 1992.

[34] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, P. Sommerlad, and M. Stal, *Pattern-oriented software architecture, volume 1: A system of patterns*, John Wiley & Sons, 1996.

[35] F. W. Calliss, *Problems with automatic restructurers*, SIGPLAN Notices, 23 (1988), pp. 13–21.

[36] case/4/0. `http://www.microtool.de/case40/en/index.asp`.

[37] S. Ceri, P. Fraternali, and A. Bongio, *Web modeling language (WebML): A modeling language for designing Web sites*, in WWW9: Proceedings of the 9^{th} International World Wide Web Conference on Computer Networks, North-Holland Publishing Co., 2000, pp. 137–157.

[38] C. Chambers, B. Harrison, and J. Vlissides, *A debate on language and tool support for design patterns*, in POPL '00: Proceedings of the 27^{th} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 2000, pp. 277–289.

[39] A. Cherchago and R. Heckel, *Specification matching of web services using conditional graph transformation rules*, in ICGT '04: Proceedings of the 2^{nd} International Conference on Graph Transformation, H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, eds., vol. 3256 of Lecture Notes in Computer Science, Springer, 2004, pp. 304–318.

[40] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, *NuSMV: A new symbolic model verifier*, in CAV '99: Proceedings of the 11^{th} Conference on Computer-Aided Verification, N. Halbwachs and D. Peled, eds., no. 1633 in Lecture Notes in Computer Science, Springer, 1999, pp. 495–499.

[41] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*, The MIT Press, 2000.

[42] E. M. Clarke, J. M. Wing, R. Alur, R. Cleaveland, D. Dill, A. Emerson, S. Garland, S. German, J. Guttag, A. Hall, T. Henzinger, G. Holzmann, C. Jones, R. Kurshan, N. Leveson, K. McMillan, J. Moore, D. Peled, A. Pnueli, J. Rushby, N. Shankar, J. Sifakis, P. Sistla, B. Steffen, P. Wolper, J. Woodcock, and P. Zave, *Formal methods: State of the art and future directions*, ACM Computing Surveys, 28 (1996), pp. 626–643.

[43] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada, *Maude: Specification and programming in rewriting logic*, Theoretical Computer Science, 285 (2002), pp. 187–243.

[44] J. S. Cuadrado, J. G. Molina, and M. M. Tortosa, *RubyTL: A practical, extensible transformation language*, in ECMDA-FA '06: Proceedings of the 2^{nd} European Conference on Model Driven Architecture - Foundations and Applications, A. Rensink and J. Warmer, eds., vol. 4066 of Lecture Notes in Computer Science, Springer, 2006, pp. 158–172.

[45] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Attributed graph transformation with node type inheritance*, Theoretical Computer Science, 376 (2007), pp. 139–163.

[46] J. de Lara, E. Guerra, and P. Bottoni, *Triple patterns: Compact specification for the generation of operational triple graph grammar rules*, in GTVMT '07: Proceedings of the 6^{th} International Workshop on Graph Transformation and Visual Modeling Techniques (Satellite Event of ETAPS 2007), vol. 6 of Electronic Communications of the EASST, 2007.

[47] J. de Lara, E. Guerra, and H. Vangheluwe, *Meta-modelling, graph transformation and model checking for the analysis of hybrid systems*, in AGTIVE '03: Proceedings of the 2^{nd} International Workshop on Applications of Graph Transformations with Industrial Relevance, J. L. Pfaltz, M. Nagl, and B. Böhlen, eds., vol. 3062 of Lecture Notes in Computer Science, Springer, 2003, pp. 292–298.

[48] ——, *A multi-view component modelling language for systems design: Checking consistency and timing constraints*, in VMSIS '05: Proceedings of the 2005 Workshop on Visual Modeling for Software Intensive Systems (Satellite Event of VLHCC 2005), 2005, pp. 27–34.

[49] J. de Lara and G. Taentzer, *Automated model transformation and its validation using AToM^3 and AGG*, in DIAGRAMS '04: Proceedings of the 3^{rd} International Conference on Diagrammatic Representation and Inference, A. F. Blackwell, K. Marriott, and A. Shimojima, eds., vol. 2980 of Lecture Notes in Computer Science, Springer, 2004, pp. 182–198.

[50] J. de Lara and H. Vangheluwe, *Computer aided multi-paradigm modelling to process Petri-nets and statecharts*, in ICGT '02: Proceedings of the 1^{st} International Conference on Graph Transformation, Springer-Verlag, 2002, pp. 239–253.

[51] ——, *AToM^3: A tool for multi-formalism and meta-modelling*, in FASE '02: Proceedings of the 5^{th} International Conference on Fundamental Approaches to Software Engineering, Springer-Verlag, 2002, pp. 174–188.

[52] ——, *Defining visual notations and their manipulation through meta-modeling and graph transformation*, Journal of Visual Languages and Computing, 15 (2004), pp. 309–330.

[53] P. Díaz, I. Aedo, and F. Panetsos, *Modeling the dynamic behavior of hypermedia applications*, IEEE Transactions on Software Engineering, 27 (2001), pp. 550–572.

[54] P. Díaz, S. Montero, and I. Aedo, *Modelling hypermedia and web applications: The Ariadne Development Method*, Information Systems, 30 (2005), pp. 649–673.

[55] DSLTools. `http://msdn.microsoft.com/vstudio/DSLTools`.

[56] F. Durán and A. Vallecillo, *Formalizing ODP enterprise specifications in Maude*, Computer Standards & Interfaces, 25 (2003), pp. 83–102.

[57] S. Easterbrook and B. Nuseibeh, *Using viewpoints for inconsistency management*, Software Engineering Journal, 11 (1996), pp. 31–43.

[58] N. V. Eetvelde and D. Janssens, *Extending graph rewriting for refactoring*, in ICGT '04: Proceedings of the 2^{nd} International Conference on Graph Transformation, H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, eds., vol. 3256 of Lecture Notes in Computer Science, Springer, 2004, pp. 399–415.

[59] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay, *Termination criteria for model transformation*, in FASE '05: Proceedings of the 8^{th} International Conference on Fundamental Approaches to Software Engineering, M. Cerioli, ed., vol. 3442 of Lecture Notes in Computer Science, Springer-Verlag, 2005, pp. 49–63.

[60] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Formal integration of inheritance with typed attributed graph transformation for efficient vl definition and model manipulation*, in VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages - Human Centric Computing, IEEE Computer Society, 2005, pp. 71–78.

[61] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of algebraic graph transformation (Monographs in theoretical computer science. An EATCS series)*, Springer-Verlag New York, Inc., 2006.

[62] H. Ehrig, U. Prange, and G. Taentzer, *Fundamental theory for typed attributed graph transformation*, in ICGT '04: Proceedings of the 2^{nd} International Conference on Graph Transformation, H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, eds., vol. 3256 of Lecture Notes in Computer Science, Springer, 2004, pp. 161–177.

[63] K. Ehrig, C. Ermel, S. Hänsgen, and G. Taentzer, *Generation of visual editors as Eclipse plug-ins*, in ASE '05: Proceedings of the 20^{th} IEEE International Conference on Automated Software Engineering, ACM Press, 2005, pp. 134–143.

[64] K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, G. Taentzer, D. Varró, and S. V. Gyapay, *Model transformation by graph transformation: A comparative study*, in MTiP '05: Proceedings of the International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005), 2005.

[65] EMFQuery. `http://www.eclipse.org/emft/projects/query/#query`.

[66] C. Ermel, K. Holscher, S. Kuske, and P. Ziemann, *Animated simulation of integrated UML behavioral models based on graph transformation*, in VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages - Human Centric Computing, IEEE Computer Society, 2005, pp. 125–133.

[67] J. M. Favre, *Towards a basic theory to model model driven engineering*, in WISME '04: Proceedings of the 3^{rd} Workshop on Software Model Engineering, 2004.

[68] N. E. Fenton, *Software metrics: A rigorous and practical approach (2^{nd} edition)*, International Thomson Computer Press, 1996.

[69] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, *Viewpoints: A framework for integrating multiple perspectives in system development*, International Journal of Software Engineering and Knowledge Engineering, 2 (1992), pp. 31–57.

[70] T. Fischer, J. Niere, L. Torunski, and A. Zündorf, *Story Diagrams: A new graph rewrite language based on the Unified Modeling Language and Java*, in TAGT '98: Proceedings of the 6^{th} International Workshop on Theory and Application of Graph Transformations, vol. 1764 of Lecture Notes in Computer Science, Springer-Verlag, 2000, pp. 296–309.

[71] P. A. Fishwick, *A simulation environment for multimodeling*, Discrete event dynamic systems: Theory and applications, 3 (1993), pp. 151–171.

[72] FormalCheck, *Users guide, v2.1.* Bell Labs Design Automation, Lucent Technologies, 1998.

[73] M. Fowler, *Analysis patterns: Reusable objects models*, Addison-Wesley Longman Publishing Co., Inc., 1997.

[74] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the design of existing code*, Addison-Wesley Professional, 1999.

[75] R. B. France, M. P. Evett, and E. Grant, *Towards semantic-based object-oriented CASE tools*, in ASE '97: Proceedings of the 12^{th} IEEE International Conference on Automated Software Engineering, IEEE Computer Society, 1997, pp. 295–296.

[76] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley, 1995.

[77] F. García, M. F. Bertoa, C. Calero, A. Vallecillo, F. Ruiz, M. Piattini, and M. Genero, *Towards a consistent terminology for software measurement*, Information & Software Technology, 48 (2006), pp. 631–644.

[78] GMF, *The Eclipse Graphical Modeling Framework*. `http://www.eclipse.org/gmf`.

[79] M. Goedicke, B. E. Enders, T. Meyer, and G. Taentzer, *Towards integrating multiple perspectives by distributed graph transformation*, in AGTIVE '99: Proceedings of the 1^{st} International Workshop on Applications of Graph Transformations with Industrial Relevance, vol. 1779 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 369–377.

[80] GRADE. `http://www.gradetools.com/`.

[81] J. Grundy, R. Mugridge, J. Hosking, and P. Kendall, *A visual language and environment for EDI message translation*, in HCC '01: Proceedings of the 2001 IEEE Symposia on Human Centric Computing Languages and Environments, IEEE Computer Society, 2001, pp. 330–331.

[82] J. C. Grundy, W. B. Mugridge, and J. G. Hosking, *Visual specification of multiview visual environments*, in VL '98: Proceedings of the 1998 IEEE Symposium on Visual Languages, IEEE Computer Society, 1998, pp. 236–243.

[83] E. Guerra and J. de Lara, *A framework for the verification of UML models. Examples using Petri nets*, in JISBD, 2003, pp. 325–334.

[84] ——, *Attributed typed triple graph transformation with inheritance in the double pushout approach*, Tech. Report UC3M-TR-CS-06-01 (61 páginas), Universidad Carlos III de Madrid, 2006.

[85] ——, *Adding recursion to graph transformation*, in GTVMT '07: Proceedings of the 6^{th} International Workshop on Graph Transformation and Visual Modeling Techniques (Satellite Event of ETAPS 2007), vol. 6 of Electronic Communications of the EASST, 2007.

[86] ——, *Event-driven grammars: Relating abstract and concrete levels of visual languages*, Journal on Software and Systems Modeling, special section on ICGT'04, (2007), pp. 317–347.

[87] ——, *Visual languages for interactive computing: Definitions and formalizations*, Idea Group Publishers, 2007, ch. Meta-modelling and graph transformation for the definition of multi-view visual languages.

[88] E. Guerra, J. de Lara, and A. Malizia, *Model-driven development of digital libraries: Validation, analysis and code generation*, in WEBIST '07: Proceedings of the 3^{rd} International Conference on Web Systems and Technologies, Lecture Notes in Business Information Processing, Springer, 2007.

[89] E. Guerra, D. Sanz, P. Díaz, and I. Aedo, *A transformation-driven approach to the verification of security policies in web designs*, in ICWE '07: Proceedings of the 7^{th} International Conference on Web Engineering, L. Baresi, P. Fraternali, and G.-J. Houben, eds., vol. 4607 of Lecture Notes in Computer Science, Springer, 2007, pp. 269–284.

[90] J. Hahn and J. Kim, *Why are some diagrams easier to work with? Effects of diagrammatic representation on the cognitive integration process of systems analysis and design*, ACM Transactions on Computer-Human Interaction, 6 (1999), pp. 181–213.

[91] J. H. Hausmann, R. Heckel, and S. Sauer, *Extended model relations with graphical consistency conditions*, in Research Report 2002:06 of Blekinge Institute of Technology. Workshop on Consistency Problems in UML-based Software Development (Satellite Event of UML'02), L. Kuzniarz, G. Reggio, J. L. Sourrouille, and Z. Huzar, eds., 2002, pp. 61–74.

[92] R. Heckel, J. M. Küster, and G. Taentzer, *Confluence of typed attributed graph transformation systems*, in ICGT '02: Proceedings of the 1^{st} International Conference on Graph Transformation, Springer-Verlag, 2002, pp. 161–176.

[93] R. Heckel and A. Wagner, *Ensuring consistency of conditional graph rewriting - A constructive approach*, Electronic Notes in Theoretical Computer Science, 2 (1995).

[94] G. J. Holzmann, *The model checker SPIN*, Software Engineering, 23 (1997), pp. 279–295.

[95] IEEE/Std-1471-2000, *IEEE recommended practice for architectural description of software-intensive systems*, 2000.

[96] IntelliJIDEA. `http://www.jetbrains.com/idea`.

[97] ISO/IEC-15939, *2002 software engineering – software measurement process.*

[98] ISO/IEC-9126, *1991 software engineering – product quality.*

[99] I. Ivkovic and K. Kontogiannis, *A framework for software architecture refactoring using model transformations and semantic annotations*, in CSMR '06: Proceedings of the 10^{th} European Conference on Software Maintenance and Reengineering, IEEE Computer Society, 2006, pp. 135–144.

[100] J. Jakob, A. Königs, and A. Schürr, *Non-materialized model view specification with triple graph grammars*, in ICGT '06: Proceedings of the 3^{rd} International Conference on Graph Transformation, A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, eds., vol. 4178 of Lecture Notes in Computer Science, Springer, 2006, pp. 321–335.

[101] A. R. Jansen, K. Marriott, and B. Meyer, *Cider: A component-based toolkit for creating smart diagram environments*, in DIAGRAMS '04: Proceedings of the 3^{rd} International Conference on Diagrammatic Representation and Inference, A. F. Blackwell, K. Marriott, and A. Shimojima, eds., vol. 2980 of Lecture Notes in Computer Science, Springer, 2004, pp. 415–419.

[102] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle, *On the use of graph transformation in the formal specification of model interpreters*, Journal of Universal Computer Science, 9 (2003), pp. 1296–1321.

[103] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Norkin, *Automated support for program refactoring using invariants*, in ICSM '01: Proceedings of the 17^{th} IEEE International Conference on Software Maintenance, IEEE Computer Society, 2001, pp. 736–743.

[104] H. Kazato, M. Takaishi, T. Kobayashi, and M. Saeki, *Formalizing refactoring by using graph transformation (metrics, test, and maintenance)*, IEICE Transactions on Information and Systems, 87 (2004), pp. 855–867.

[105] J. Kerievsky, *Refactoring to patterns*, Addison-Wesley, 2004.

[106] P. Klein and A. Schürr, *Constructing SDEs with the IPSEN meta environment*, in SEE '97: Proceedings of the 8^{th} International Conference on Software Engineering Environments, IEEE Computer Society, 1997, pp. 2–10.

[107] N. Koch and G. Rossi, *Patterns for adaptive web applications*, in EuroPlop '02: Proceedings of the 7^{th} European Conference on Pattern Languages of Programs, 2002.

[108] A. Königs, *Model transformation with triple graph grammars*, in MTiP '05: Proceedings of the International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005), 2005.

[109] ——, *Model integration and transformation - A triple graph grammar-based QVT implementation*, PhD thesis, University of Technology of Darmstadt, 2007.

[110] S. Lack and P. Sobocinski, *Adhesive categories*, in FOSSACS '04: Proceedings of the 7^{th} International Conference on Foundations of Software Science and Computation Structures, I. Walukiewicz, ed., vol. 2987 of Lecture Notes in Computer Science, Springer, 2004, pp. 273–288.

[111] S. Lacoste-Julien, H. Vangheluwe, J. de Lara, and P. Mosterman, *Meta-modelling hybrid formalisms*, in IEEE International Symposium on Computer Aided Control System Design, special section on multi-paradigm modelling, 2004.

[112] L. Lambers, H. Ehrig, and F. Orejas, *Conflict detection for graph transformation with negative application conditions*, in ICGT '06: Proceedings of the 3^{rd} International Conference on Graph Transformation, A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, eds., vol. 4178 of Lecture Notes in Computer Science, Springer, 2006, pp. 61–76.

[113] R. Lämmel, *Towards generic refactoring*, in RULE '02: Proceedings of the 2002 ACM SIGPLAN workshop on Rule-based programming, ACM Press, 2002, pp. 15–28.

[114] J. A. Landay and G. Borriello, *Design patterns for ubiquitous computing*, Computer, 36 (2003), pp. 93–95.

[115] M. Lanza and S. Ducasse, *Beyond language independent object-oriented metrics: Model independent metrics*, in QAOOSE '02: Proceedings of the 6^{th} International Workshop on Quantitative Approaches in Object-Oriented Software Engineering, F. B. e Abreu, M. Piattini, G. Poels, and H. A. Sahraoui, eds., 2002, pp. 77–84.

[116] J. Larkin and H. A. Simon, *Why a diagram is (sometimes) worth 10,000 words*, Cognitive Science, 11 (1987), pp. 65–100.

[117] F. W. Lawvere and S. H. Schanuel, *Conceptual mathematics: A first introduction to categories*, Buffalo Workshop Press, 1991.

[118] A. Lédeczi, A. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, *Composing domain-specific design environments*, Computer, 34 (2001), pp. 44–51.

[119] J. Lilius and I. Porres, *vUML: A tool for verifying UML models*, in ASE '99: Proceedings of the 14^{th} IEEE International Conference on Automated Software Engineering, IEEE Computer Society, 1999, pp. 255–258.

[120] M. Lindvall, P. Donzelli, S. Asgari, and V. Basili, *Towards reusable measurement patterns*, in METRICS '05: Proceedings of the 11^{th} IEEE International Software Metrics Symposium, IEEE Computer Society, 2005, pp. 21–28.

[121] R. J. Machado, K. B. Lassen, S. Oliveira, M. Couto, and P. Pinto, *Execution of UML models with CPN tools for workflow requirements validation*, in 6^{th} Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, 2005, pp. 231–250.

[122] ——, *Requirements validation: Execution of UML models with CPN tools*, International Journal on Software Tools for Technology Transfer, 9 (2007), pp. 353–369.

[123] S. MacLane, *Categories for the working mathematician*, vol. 5, Springer-Verlag, 2^{nd} edition ed., 1998.

[124] MagicDraw. `http://www.magicdraw.com/`.

[125] MAISA, *Metrics for Analysis and Improvement of Software Architectures*. `http://www.cs.helsinki.fi/group/maisa/`.

[126] A. Malizia, E. Guerra, and J. de Lara, *Model-driven development of digital libraries: Generating the user interface*, in MDDAUI '06: Proceedings of the 2^{nd} International Workshop on Model Driven Development of Advanced User Interfaces (Satellite Event of MoDELS 2006), vol. 214 of CEUR Workshop Proceedings, CEUR-WS.org, 2006.

[127] Z. Manna and A. Pnueli, *A hierarchy of temporal properties (invited paper, 1989)*, in PODC '90: Proceedings of the 9^{th} ACM Symposium on Principles of Distributed Computing, ACM Press, 1990, pp. 377–410.

[128] M. A. Martín and L. Olsina, *Towards an ontology for software metrics and indicators as the foundation for a cataloging web system*, in LA-WEB '03: Proceedings of the 1^{st} Conference on Latin American Web Congress, IEEE Computer Society, 2003, pp. 103–113.

[129] T. J. McCabe, *A complexity measure*, IEEE Transactions on Software Engineering, 2 (1976), pp. 308–320.

[130] J. A. McQuillan and J. F. Power, *Towards re-usable metric definitions at the meta-level*, in PhD Workshop of the 20^{th} European Conference on Object-Oriented Programming (ECOOP 2006), 2006, pp. 3–7.

[131] MDA, *OMG's page about Model-Driven Architecture*. `http://www.omg.org/mda/`.

[132] mediniQVT. `http://www.ikv.de/mediniQVT/`.

[133] T. Mens, *On the use of graph transformations for model refactoring*, in GTTSE '05: Proceedings of the Generative and Transformational Techniques in Software Engineering, International Summer School, R. Lämmel, J. Saraiva, and J. Visser, eds., vol. 4143 of Lecture Notes in Computer Science, Springer, 2006, pp. 219–257.

[134] T. Mens and P. V. Gorp, *A taxonomy of model transformation*, Electronic Notes in Theoretical Computer Science, 152 (2006), pp. 125–142.

[135] T. Mens and M. Lanza, *A graph-based metamodel for object-oriented software metrics*, Electronic Notes in Theoretical Computer Science, 72 (2002).

[136] MetaEdit. `http://www.metacase.com/mep/`.

[137] MetricView/MetricViewEvolution. `http://www.win.tue.nl/empanada/metricview/`.

[138] M. Minas, *Concepts and realization of a diagram editor generator based on hypergraph transformation*, Science of Computer Programming, 44 (2002), pp. 157–180.

[139] ——, *Generating visual editors based on Fujaba/MOFLON and DiaMeta*, Tech. Report tr-ri-06-275, University Paderborn, 2006.

[140] V. B. Misic and S. Moser, *From formal metamodels to metrics: An object-oriented approach*, in TOOLS '97: Proceedings of the 25^{th} Technology of Object-Oriented Languages and Systems-Tools, IEEE Computer Society, 1997, pp. 330–339.

[141] MOFLON. `http://www.moflon.org`.

[142] MOLA, *MOdel transformation LAnguage*. `http://mola.mii.lu.lv/`.

[143] MTF, *Model Transformation Framework*. `http://www.alphaworks.ibm.com/tech/mtf`.

[144] M. J. Munro, *Product metrics for automatic identification of "bad smell" design problems in Java source-code*, in METRICS '05: Proceedings of the 11^{th} IEEE International Software Metrics Symposium, IEEE Computer Society, 2005, pp. 15–23.

[145] T. Murata, *Petri nets: Properties, analysis and applications*, Proceedings of the IEEE, 77 (1989), pp. 541–580.

[146] Objecteering. `http://www.objecteering.com/`.

[147] OMEGA. `http://www-omega.imag.fr/index.php`.

[148] OpenArchitectureWare. `http://www.openarchitectureware.org/`.

[149] OptimalJ. `http://www.compuware.com/products/optimalj/`.

[150] F. Pérez, J. de Lara, and E. Guerra, *Domain specific languages with graphical and textual views*, in AGTIVE '07: Proceedings of the 3^{rd} International Workshop on Applications of Graph Transformations with Industrial Relevance, 2007, pp. 79–94.

[151] D. Plump, *On termination of graph rewriting*, in WG '95: Proceedings of the 21^{st} International Workshop on Graph-Theoretic Concepts in Computer Science, Springer-Verlag, 1995, pp. 88–100.

[152] R. Pohjonen and J.-P. Tolvanen, *Automated production of family members: Lessons learned*, in Proceedings of the International Workshop on Product Line Engineering The Early Steps: Planning, Modeling, and Managing, 2002, pp. 49–57.

[153] Poseidon. `http://www.gentleware.com/poseidon.html`.

[154] R. S. Pressman, *Ingeniería del software 6/E*, McGraw-Hill, 2005.

[155] QVT, *OMG's specification for Queries/Views/Transformations*. `http://www.omg.org/docs/ptc/05-11-01.pdf`.

[156] A. Rensink, *Representing first-order logic using graphs*, in ICGT '04: Proceedings of the 2^{nd} International Conference on Graph Transformation, H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, eds., vol. 3256 of Lecture Notes in Computer Science, Springer, 2004, pp. 319–335.

[157] D. Roberts, J. Brant, and R. Johnson, *A refactoring tool for Smalltalk*, Theory and Practice of Object Systems, 3 (1997), pp. 253–263.

[158] F. Rosselló and G. Valiente, *Analysis of metabolic pathways by graph transformation*, in ICGT '04: Proceedings of the 2^{nd} International Conference on Graph Transformation, H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, eds., vol. 3256 of Lecture Notes in Computer Science, Springer, 2004, pp. 70–82.

[159] G. Rozenberg, ed., *Handbook of graph grammars and computing by graph transformation: Volume I. Foundations*, World Scientific Publishing Co., Inc., 1997.

[160] R. S. Sandhu, *A perspective on graphs and access control models*, in ICGT '04: Proceedings of the 2^{nd} International Conference on Graph Transformation, H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, eds., vol. 3256 of Lecture Notes in Computer Science, Springer, 2004, pp. 2–12.

[161] T. Schäfer, A. Knapp, and S. Merz, *Model checking UML state machines and collaborations*, Electronic Notes in Theoretical Computer Science, 55 (2001).

[162] A. Schürr, *Specification of graph translators with triple graph grammars*, in WG '94: Proceedings of the 20^{th} International Workshop on Graph-Theoretic Concepts in Computer Science, Springer-Verlag, 1995, pp. 151–163.

[163] D. Schwabe and G. Rossi, *An object oriented approach to web-based applications design*, Theory and Practice of Object Systems, 4 (1998), pp. 207–225.

[164] SDMetrics. `http://www.sdmetrics.com`.

[165] SES/workbench. `http://www.mmsolutions.com/english/workbench.htm`.

[166] R. Sierra, *Tesis doctorales y trabajos de investigación científica (3^a ed.)*, Paraninfo, 1994.

[167] F. Simon, S. Löffler, and C. Lewerentz, *Distance based cohesion measuring*, in FESMA '99: Proceedings of the 2^{nd} European Software Measurement on Computer Networks.

[168] SmartQVT. `http://smartqvt.elibel.tm.fr/`.

[169] O. Sokolsky, I. Lee, and H. Ben-Abdallah, *Specification and analysis of real-time systems with PARAGON*, in Annals of Software Engineering, vol. 7, Springer, 1999, pp. 211–234.

[170] J. M. Spivey, *The Z notation: A reference manual*, Prentice-Hall, Inc., 1989.

[171] SPQR/20, *User manual*. Software Productivity Research Inc., 1995.

[172] J. Sprinkle and G. Karsai, *A domain-specific visual language for domain model evolution*, Journal of Visual Languages and Computing, 15 (2004), pp. 291–307.

[173] D. Stein, S. Hanenberg, and R. Unland, *Query models*, in UML '04: Proceedings of the 7^{th} International Conference on the Unified Modelling Language, T. Baar, A. Strohmeier, A. M. D. Moreira, and S. J. Mellor, eds., vol. 3273 of Lecture Notes in Computer Science, Springer, 2004, pp. 98–112.

[174] StP, *Software through Pictures*. `http://www.aonix.com/stp_uml.html`.

[175] T. T. Sunetnanta and A. Finkelstein, *Automated consistency checking for multiperspective software applications*, in Workshop on Advanced Separation of Concerns (Satellite Event at ICSE'01), 2001.

[176] J. G. Süss, A. Leicher, and S. Busse, *OCLPrime - Environment and language for model query, views and transformations*, Electronic Notes in Theoretical Computer Science, 102 (2004), pp. 133–153.

[177] G. Taentzer, *Parallel and distributed graph transformation. Formal description and application to communication-based systems*, PhD thesis, TU Berlin, 1996.

[178] G. Taentzer and A. Rensink, *Ensuring structural constraints in graph-based models with type inheritance*, in FASE '05: Proceedings of the 8^{th} International Conference on Fundamental Approaches to Software Engineering, M. Cerioli, ed., vol. 3442 of Lecture Notes in Computer Science, Springer-Verlag, 2005, pp. 64–79.

[179] Together. `http://www.borland.com/us/products/together`.

[180] T. Tourwé and T. Mens, *Identifying refactoring opportunities using logic meta programming*, in CSMR '03: Proceedings of the 7^{th} European Conference on Software Maintenance and Reengineering, IEEE Computer Society, 2003, pp. 91–100.

[181] C. Tsalidis, D. Christodoulakis, and D. Maritsas, *ATHENA: A software measurement and metrics environment*, Journal of Software Maintenance, 4 (1992), pp. 61–81.

[182] UML, *The Unified Modeling Language*. `http://www.omg.org/UML`.

[183] C. Nentwich, W. Emmerich, A. Finkelstein, and E. Ellmer, *Flexible consistency checking*, ACM Transactions on Software Engineering and Methodology, 12 (2003), pp. 28–63.

[184] H. Vangheluwe and J. de Lara, *Computer automated multi-paradigm modelling for analysis and design of traffic networks*, in WSC '04: Proceedings of the 36^{th} Winter Simulation Conference, 2004, pp. 249–258.

[185] H. Vangheluwe, J. de Lara, and P. J. Mosterman, *An introduction to multi-paradigm modelling and simulation*, in AI Simulation and Planning AIS-2002, 2002, pp. 9–20.

[186] D. Varró, G. Varró, and A. Pataricza, *Designing the automatic transformation of visual languages*, Science of Computer Programming, 44 (2002), pp. 205–227.

[187] P. J. Vázquez, M. N. Moreno, and F. J. García, *Métricas orientadas a objetos*, Tech. Report DPTOIA-IT-2001-02, Universidad de Salamanca, 2001.

[188] M. Völter and T. Stahl, *Model-driven software development: Technology, engineering, management*, Willey, 2006.

[189] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting your models ready for MDA (2^{nd} edition)*, Addison-Wesley Longman Publishing Co., Inc., 2003.

[190] M. Weiss, *A pattern language for motivating the use of agents*, in AOIS '03: Proceedings of the Agent-Oriented Information Systems, 5^{th} International Bi-Conference Workshop, P. Giorgini, B. Henderson-Sellers, and M. Winikoff, eds., vol. 3030 of Lecture Notes in Computer Science, Springer, 2003, pp. 142–157.

[191] S. A. Whitmire, *Object oriented design measurement*, John Wiley & Sons, Inc., 1997.

[192] F. Xie, V. Levin, and J. C. Browne, *ObjectCheck: A model checking tool for executable object-oriented software system designs*, in FASE '02: Proceedings of the 5^{th} International Conference on Fundamental Approaches to Software Engineering, Springer-Verlag, 2002, pp. 331–335.

[193] XUL, *XML User Interface Language*. `http://www.mozilla.org/projects/xul`.

[194] B. P. Zeigler, T. G. Kim, and H. Praehofer, *Theory of modeling and simulation*, Academic Press, Inc., 2000.

[195] N. Zhu, J. Grundy, J. Hosking, N. Liu, S. Cao, and A. Mehra, *Pounamu: A meta-tool for exploratory domain-specific visual language tool development*, Journal of Systems and Software, 80 (2007), pp. 1390–1407.